CHAPTER 20

Lockouts and Two-Factor Sign-Ins

In this chapter, I describe how account lockouts are supported and explain the mechanisms behind two-factor authentication. Table 20-1 puts these features into context.

Table 20-1. *Putting Lockouts and Two-Factor Sign-Ins in Context*

Question	Answer
What are they?	Lockouts prevent a user from signing in after a specified number of failed attempts. Two-factor sign-ins require the user to provide additional credentials during the sign-in process.
Why are they useful?	Both features are intended to improve security. Lockouts prevent attackers from repeatedly trying to guess passwords. Two-factor authentication requires attackers to have access to the user's additional credentials.
How are they used?	Both features are implemented using optional user store interfaces.
Are there any pitfalls or limitations?	Users can find both features frustrating, and it is important to make the workflows as clear and as easy to use as possible.
Are there any alternatives?	The best alternative is to use external authentication, where the process of identifying users is delegated to a third party. See Chapter 22 for details.

Table 20-2 summarizes the chapter.

Table 20-2. *Chapter Summary*

Problem	Solution	Listing
Support account lockouts	Implement the `IUserLockoutStore<T>` interface and use the user manager methods for management.	2–5, 7, 8
Configure lockouts	Use the options pattern.	6
Restrict access to confirmed accounts	Create an implementation of the `IUserConfirmation<T>` interface.	9–12
Support two-factor authentication	Implement the `IUserTwoFactorStore<T>` interface and use the methods defined by the sign-in manager to sign users in.	13–29

Preparing for This Chapter

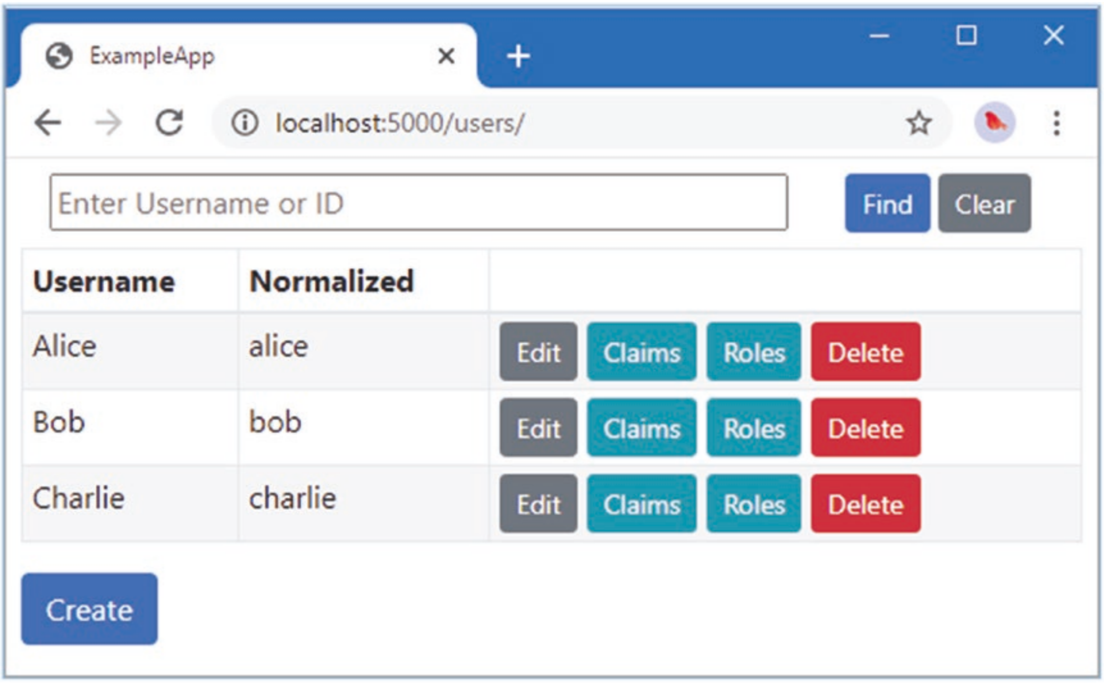
This chapter uses the `ExampleApp` project from Chapter 19. No changes are required to prepare for this chapter. Open a new command prompt, navigate to the `ExampleApp` folder, and run the command shown in Listing 20-1 to start ASP.NET Core.

■ **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 20-1. 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 20-1. 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 20-1. *Running the example application*

Enabling Lockouts

When account lockouts are used, Identity keeps track of how many failed attempts to sign in are made. If there are too many failed attempts in a short period, the account will be locked out, meaning that the user won't be signed in even if the correct password is provided. Lockouts are a useful feature for preventing attackers from repeatedly trying to guess a user's passwords, although care must be taken because lockouts can also be frustrating to legitimate users who can't remember their password. (It is for this reason that lockouts should be used with a relaxed password validation policy, as explained in Chapter 19.)

Extending the User Class

The first step is to extend the user class to add properties for keeping track of whether a user account supports lockouts and the current lockout status, as shown in Listing 20-2.

Listing 20-2. 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; }
    }
}
```

CanUserBeLockedout will be used to determine if a specific user can be locked out. The FailedSignInCount property will record how many failed sign-in attempts have been made. The LockoutEnd property will keep track of when a lockout will end and returns null when the user is not locked out.

Enabling Lockouts in the User Store

The next step is to extend the user store so that it implements the IUserLockoutStore<T> interface, where T is the user class. This interface defines the methods described in Table 20-3, which are used to keep track of failed sign-in attempts and whether an account is locked. As with the methods defined by other user store interfaces, the token parameter in Table 20-3 is a CancellationToken object that is used to receive notifications when an asynchronous operation is canceled.

Table 20-3. *The IUserLockoutStore<T> Methods*

Name	Description
`GetLockoutEnabledAsync(user, token)`	This method is used to determine if the specifier user is subject to lockouts.
`SetLockoutEnabledAsync(user, enabled, token)`	This method sets whether the specified user is subject to lockouts.
`IncrementAccessFailedCountAsync(user, token)`	This method increments the number of failed sign-in attempts for the specified user.
`GetAccessFailedCountAsync(user, token)`	This method returns the number of failed sign-in attempts for the specified user.
`ResetAccessFailedCountAsync(user, token)`	This method resets the number of failed sign-in attempts for the specified user.
`SetLockoutEndDateAsync(user, end, token)`	This method sets the end of the lockout period for the specified user, expressed as a DateTimeOffset value.
`GetLockoutEndDateAsync(user, token)`	This method gets the end of the lockout period for the specified user, expressed as a DateTimeOffset value.

The user store only has to keep track of the lockout data and is not responsible for interpreting the data or enforcing lockouts. Add a class file named UserStoreLockouts.cs to the ExmapleApp/Identity/Store folder and use it to define the partial class shown in Listing 20-3.

Listing 20-3. The Contents of the UserStoreLockouts.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {
    public partial class UserStore : IUserLockoutStore<AppUser> {
        public Task SetLockoutEnabledAsync(AppUser user, bool enabled,
                CancellationToken token) {
            user.CanUserBeLockedout = enabled;
            return Task.CompletedTask;
        }

        public Task<bool> GetLockoutEnabledAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.CanUserBeLockedout);

        public Task<int> GetAccessFailedCountAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.FailedSignInCount);

        public Task<int> IncrementAccessFailedCountAsync(AppUser user,
            CancellationToken token) => Task.FromResult( ++user.FailedSignInCount);

        public Task ResetAccessFailedCountAsync(AppUser user,
            CancellationToken token) {
                user.FailedSignInCount = 0;
                return Task.CompletedTask;
        }

        public Task SetLockoutEndDateAsync(AppUser user, DateTimeOffset? lockoutEnd,
                CancellationToken token) {
            user.LockoutEnd = lockoutEnd;
            return Task.CompletedTask;
        }

        public Task<DateTimeOffset?> GetLockoutEndDateAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.LockoutEnd);
    }
}
```

The interface implementation follows the pattern shown in earlier chapters and maps the methods described in Table 20-3 onto the properties added to the AppUser class.

Managing Account Lockouts

It is important to provide the means to end lockouts that have been triggered by accident. Applications that use lockouts but are deployed without manual overrides typically run into problems when a VIP user, such as the company CEO, locks themselves out of the application. Such users do not appreciate an explanation of your security policy and do not expect to wait until the lockout ends naturally. The UserManager<T> class provides a set of members for working with lockouts, as described in Table 20-4.

Table 20-4. *The UserManager<T> Members for Lockouts*

Name	Description
`SupportsUserLockout`	This property returns `true` if the user store implements the `IUserLockoutStore<T>` interface.
`IsLockedOutAsync(user)`	This method returns `true` if the specified user is locked out, which it does by calling the store's `GetLockoutEnabledAsync` method to see if the user is subject to lockouts and then the store's `GetLockoutEndDateAsync` method to see if there is a lockout in place.
`SetLockoutEnabledAsync (user, enabled)`	This method calls the store's `SetLockoutEnabledAsync` method for the specified user and then performs user validation, updates the normalized username and email address, and then applies the user manager's update sequence.
`GetLockoutEnabledAsync(user)`	This method calls the store's `GetLockoutEnabledAsync` method for the specified user.
`GetLockoutEndDateAsync(user)`	This method calls the store's `GetLockoutEndDateAsync` method for the specified user.
`SetLockoutEndDateAsync(user, end)`	This method calls the store's `SetLockoutEndDateAsync` method for the specified user and then performs user validation, updates the normalized username and email address, and applies the user manager's update sequence.
`AccessFailedAsync(user)`	This method increments the failed sign-in counter for the specified user by calling the store's `IncrementAccessFailedCountAsync` method. If the number of failed attempts exceeds the configuration setting (described in the next section), the store's `SetLockoutEndDateAsync` method is called to lock the account, after which user validation is performed and the user manager's update sequence is applied. The number of failed attempts is reset when the account is locked.
`GetAccessFailedCountAsync(user)`	This method returns the number of failed attempts for the specified user by calling the user store's `GetAccessFailedCountAsync` method.
`ResetAccessFailedCountAsync(user)`	This method calls the store's `ResetAccessFailedCountAsync` method, performs user validation, and then applies the user manager's update sequence.

For the most part, the methods in Table 20-4 map directly onto the methods implemented by the user store. The exception is the `AccessFailedAsync` method, which increments the failed sign-in counter until it reaches a configured limit, after which the account is put into lockout.

Add a Razor Page named `UserLockouts.cshtml` to the `ExampleApp/Pages/Store` folder, with the contents shown in Listing 20-4.

Listing 20-4. The Contents of the UserLockouts.cshtml File in the Pages/Store Folder

```
@page "/users/lockout"
@model ExampleApp.Pages.Store.UserLockoutsModel

<h4 class="bg-primary text-white text-center p-2">User Lockouts</h4>
```

```
<div class="m-2">
    <table class="table table-sm table-striped">
        <thead><tr><th>Username</th><th>Lockout</th><th/></tr></thead>
        <tbody>
            @foreach (AppUser user in Model.Users) {
                <tr>
                    <td>@user.UserName</td>
                    <td>@(await Model.GetLockoutStatus(user))</td>
                    <td>
                        <form method="post">
                            <input type="hidden" name="id" value="@user.Id" />
                            @if (await Model.UserManager.IsLockedOutAsync(user)) {
                                <button class="btn btn-sm btn-secondary"
                                         type="submit">
                                     Unlock
                                </button>
                            } else {
                                <span class="mx-1">
                                    <input type="number" name="mins" value="10" />
                                    mins
                                </span>
                                <button class="btn btn-sm btn-danger" type="submit">
                                    Lock
                                </button>
                            }
                        </form>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

The content presents a list of users, with buttons that lock and unlock accounts. To define the page model, add the code shown in Listing 20-5 to the `UserLockouts.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 20-5. The Contents of the UserLockouts.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;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExampleApp.Pages.Store {

    public class UserLockoutsModel : PageModel {
        public UserLockoutsModel(UserManager<AppUser> manager)
            => UserManager = manager;

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

        public IEnumerable<AppUser> Users => UserManager.Users
             .OrderByDescending(u => UserManager.IsLockedOutAsync(u).Result)
             .ThenBy(u => u.UserName);

        public async Task<string> GetLockoutStatus(AppUser user) {
            if (await UserManager.IsLockedOutAsync(user)) {
                TimeSpan remaining = (await UserManager.GetLockoutEndDateAsync(user))
                    .GetValueOrDefault().Subtract(DateTimeOffset.Now);
                return $"Locked Out ({ remaining.Minutes } mins "
                    + $"{ remaining.Seconds} secs remaining)";
            }
            return "(No Lockout)";
        }

        public async Task<IActionResult> OnPost(string id, int mins) {
            await UserManager.SetLockoutEndDateAsync((await
                UserManager.FindByIdAsync(id)), DateTimeOffset.Now.AddMinutes(mins));
            return RedirectToPage();
        }
    }
}
```

The page model class performs a LINQ query to provide the view with users sorted by their lockout status and defines a POST handler method that locks or unlocks accounts by calling the SetLockoutEndDateAsync method. When an account is unlocked, the end of the lockout period is set to the current time. When an account is locked, the end of the lockout period is set to the number of minutes into the future specified in the HTTP request.

Configuring Lockouts

The options pattern is used to configure the lockout feature. The IdentityOptions class that is used to configure Identity defines a Lockout property that returns a LockoutOptions object. The LockoutOptions class defines the properties described in Table 20-5.

Table 20-5. *The LockoutOptions Properties*

Name	Description
AllowedForNewUsers	This property specifies whether newly created user accounts will be subject to lockouts. The default value is true.
MaxFailedAccessAttempts	This property specifies how many failed attempts are allowed before a lockout. The default value is 5.
DefaultLockoutTimeSpan	This property specifies the default lockout duration, expressed as a TimeSpan. The default value is 5 minutes.

In Listing 20-6, I have used the options pattern to reduce the number of failed sign-in attempts that trigger a lockout.

Listing 20-6. Configuring Lockouts 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;
})
.AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
.AddTokenProvider<PhoneConfirmationTokenGenerator>
    (TokenOptions.DefaultPhoneProvider)
.AddSignInManager()
.AddRoles<AppRole>();
...
```

Displaying a Lockout Notification

The final step is to make it obvious to the user when a lockout has been applied to their account. Not all applications provide this notification, but it offers the advantage of making it obvious to the user that further attempts to enter the password won't succeed. In Listing 20-7, I have modified the page model class for the SignIn Razor Page so that its view includes an error message, through which I provide details of lockouts.

Listing 20-7. Providing Lockout Details 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;
using System;

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 string Message { get; set; }

        public void OnGet(int? code) {
            if (code == StatusCodes.Status401Unauthorized) {
                Message = "401 - Challenge Response";
            } else if (code == StatusCodes.Status403Forbidden) {
                Message = "403 - Forbidden Response";
            }
            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) {
                if (result.IsLockedOut) {
                    TimeSpan remaining = (await UserManager
                        .GetLockoutEndDateAsync(user))
                        .GetValueOrDefault().Subtract(DateTimeOffset.Now);
                    Message = $"Locked Out for {remaining.Minutes} mins and"
                        + $" {remaining.Seconds} secs";
                } else {
                    Message = "Access Denied";
                }
                return Page();
            }
            return Redirect(returnUrl ?? "/signin");
        }
    }
}
```

If the result from the `SignInManager<T>.PasswordSignInAsync` method reports that the user is locked out, then I get the end time and work out how many minutes and seconds remain. In Listing 20-8, I have updated the view part of the Razor Page to display the `Message` property defined in Listing 20-7, instead of using the `Code` property to create its message strings.

Listing 20-8. Updating the View in the SignIn.cshtml File in the Pages/Store Folder

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

@if (!string.IsNullOrEmpty(Model.Message)) {
    <h3 class="bg-danger text-white text-center p-2">
        @Model.Message
    </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>
```

Restart ASP.NET Core and request `http://localhost:5000/signin`. Repeatedly enter an incorrect password and click the Sign In button. An “access denied” message is displayed for the first two failed attempts. After the third failed attempt, the account will be locked, and the remaining time displayed, as shown in Figure 20-2.

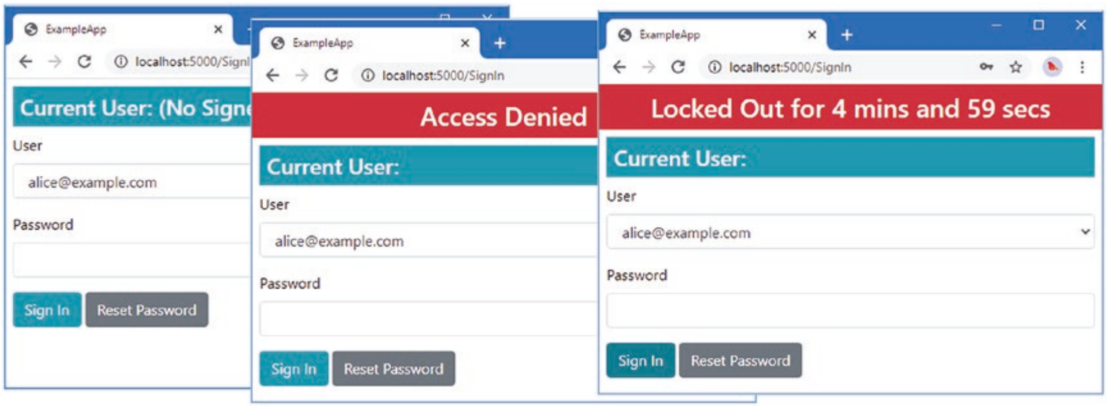


Figure 20-2. Displaying lockouts to the user

Restricting Signing In to Confirmed Accounts

The `SignInManager<T>` class performs a pre-signing check to make sure that accounts are not blocked. In addition to lockouts, accounts that have not been confirmed, or whose email or phone number has not been confirmed, can be blocked. This feature is enabled using the options pattern, where the `IdentityOptions.SignIn` property returns an instance of the `SignInOptions` class, which defines the properties described in Table 20-6.

Table 20-6. SignInOptions Properties

Name	Description
`RequireConfirmedEmail`	When this property is `true`, only accounts with confirmed email addresses can sign in. The email address confirmation is determined by calling the `UserManager<T>.IsEmailConfirmedAsync` method. The default value is `false`.
`RequireConfirmedPhoneNumber`	When this property is `true`, only accounts with confirmed phone numbers can sign in. I have not demonstrated support for phone numbers, but the process is similar to email address confirmation. The phone number confirmation is determined by calling the `UserManager<T>.IsPhoneNumberConfirmedAsync` method. The default value is `false`.
`RequireConfirmedAccount`	When this property is `true`, only confirmed accounts can sign in. Confirmation status is determined using the `IUserConfirmation<T>` interface, described after the table. The default implementation of this interface checks that the user's email address is confirmed, which means that this setting is the same as the `RequireConfirmedEmail` setting unless a custom implementation of the interface is used. The default value is `false`.

The most interesting of the properties described in Table 20-6 is `RequireConfirmedAccount`, which allows custom confirmation criteria to be defined through the `IUserConfirmation<T>` interface, which defines the method described in Table 20-7.

Table 20-7. *The IUserConfirmation<T> Method*

Name	Description
`IsConfirmedAsync(userManager, user)`	This method returns a `bool` indicating whether the specified user is confirmed.

`T` is the user class, and the `IsConfirmedAsync` method receives a `UserManager<T>` object and the user object to validate. Any confirmation criteria can be used in a custom implementation of the `IUserConfirmation<T>` interface.

Add a class file named `UserConfirmation.cs` to the `ExampleApp/Identity` folder with the contents shown in Listing 20-9.

Listing 20-9. The Contents of the UserConfirmation.cs File in the Identity Folder

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

namespace ExampleApp.Identity {
    public class UserConfirmation : IUserConfirmation<AppUser> {

        public async Task<bool> IsConfirmedAsync(UserManager<AppUser> manager,
                AppUser user) =>
            await manager.IsInRoleAsync(user, "Administrator")
                || (await manager.GetClaimsAsync(user))
                    .Any(claim => claim.Type == "UserConfirmed"
                        && string.Compare(claim.Value, "true", true) == 0);
    }
}
```

The `IsConfirmedAsync` method allows users to sign in if they have been assigned the Administrator role or have a `UserConfirmed` claim whose value is `true`. In Listing 20-10, I have extended the set of claims that can be assigned to a user to include `UserConfirmed` claims.

Listing 20-10. Adding a Claim Type in the _ClaimsRow.cshtml File in the Pages/Store Folder

```
@model (string id, Claim claim, bool newClaim)

@{ string hash = Model.claim.GetHashCode().ToString(); }

<td>
    <form method="post" id="@hash">
        <input type="hidden" name="id" value="@Model.id" />
        <input type="hidden" name="oldtype" value="@Model.claim.Type" />
        <input type="hidden" name="oldValue" value="@Model.claim.Value" />
    </form>
    <select name="type" asp-for="claim.Type" form="@hash">
        <option value="@ClaimTypes.Role">ROLE</option>
        <option value="@ClaimTypes.GivenName">GIVENNAME</option>
        <option value="@ClaimTypes.Surname">SURNAME</option>
        <option value="UserConfirmed">UserConfirmed</option>
    </select>
</td>
<td>
    <input class="w-100" name="value" value="@Model.claim.Value" form="@hash" />
</td>
<td>
    <button asp-page-handler="@(Model.newClaim ? "add" : "edit")"
        form="@hash" type="submit" class="btn btn-sm btn-info">
            @(Model.newClaim ? "Add" : "Save")
    </button>
    @if (!Model.newClaim) {
        <button asp-page-handler="delete" form="@hash" type="submit"
            class="btn btn-sm btn-danger">Delete</button>
    }
</td>
```

In Listing 20-11, I have updated the page model class for the SignIn Razor Page to check the `SignInResult.IsNotAllowed` property in the result produced by the `SignInManager`, which will be true when the user doesn't meet the confirmation criteria specified by the properties in Table 20-7.

Listing 20-11. Displaying a Message in the SignIn.cshtml.cs File in the Pages Folder

```
...
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) {
        if (result.IsLockedOut) {
            TimeSpan remaining = (await UserManager
                .GetLockoutEndDateAsync(user))
                .GetValueOrDefault().Subtract(DateTimeOffset.Now);
            Message = $"Locked Out for {remaining.Minutes} mins and"
                + $" {remaining.Seconds} secs";
        } else if (result.IsNotAllowed) {
            Message = "Sign In Not Allowed";
        } else {
            Message = "Access Denied";
        }
        return Page();
    }
    return Redirect(returnUrl ?? "/signin");
}
...
```

The final step is to register the `UserConfirmation` class as a service and enable its use through the options pattern, as shown in Listing 20-12.

Listing 20-12. Configuring the Application 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>();

    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)
    .AddSignInManager()
    .AddRoles<AppRole>();

    services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();
    services.AddSingleton<IPasswordValidator<AppUser>, PasswordValidator>();
    services.AddScoped<IUserClaimsPrincipalFactory<AppUser>,
        AppUserClaimsPrincipalFactory>();
    services.AddSingleton<IRoleValidator<AppRole>, RoleValidator>();

    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 and request `http://localhost:5000/signin`. Sign into the application as bob@example.com with the password `MySecret1$`, and you will see the error message shown in Figure 20-3, which is shown because Bob has not been assigned the `Administrator` role and does not have the `UserConfirmed` claim.

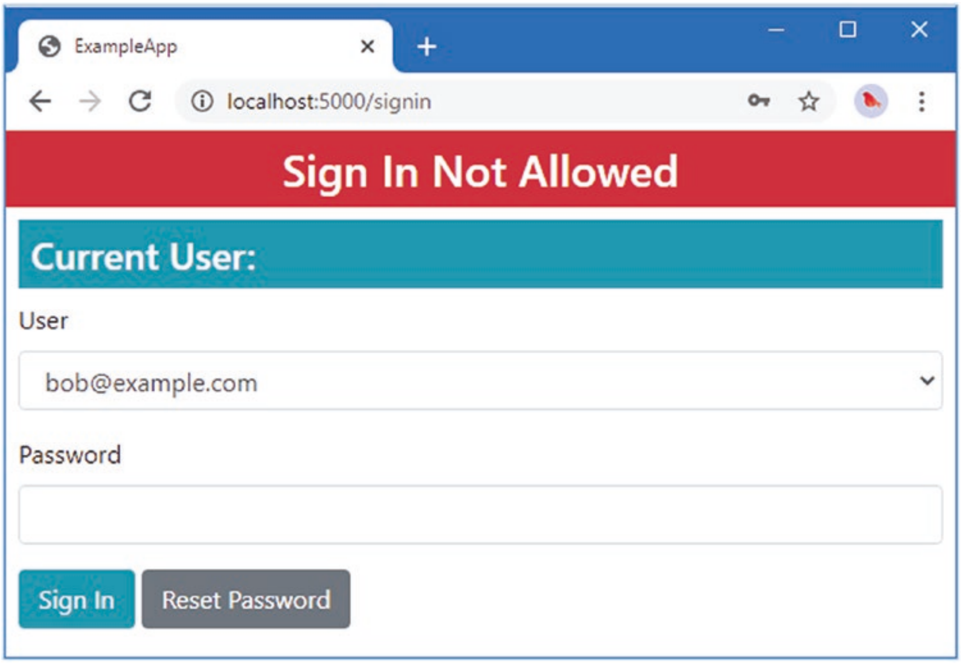


Figure 20-3. *Preventing signing in with a custom confirmation requirement*

Navigate to `http://localhost:5000/users`, click the Claims button for `Bob`, and add a `UserConfirmed` claim with a value of `true`. Return to `http://localhost:5000/signin` and repeat the sign-in process for Bob. Now that the user has been given the claim, the custom confirmation requirement will be satisfied, and sign-in will be permitted, as shown in Figure 20-4.

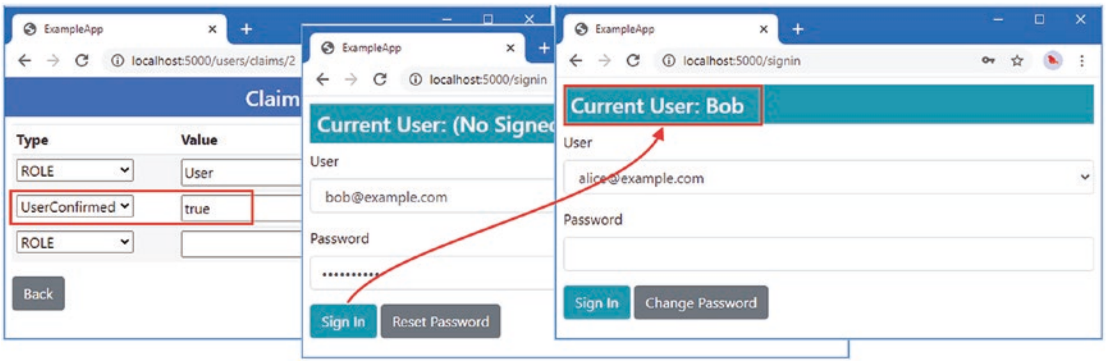


Figure 20-4. *Meeting the custom confirmation requirement*

Using Two-Factor Authentication

Two-factor authentication, often referred to as *2FA*, requires the user to provide two forms of evidence, known as *factors*, to prove their identity to the application when signing in. To make it more difficult to impersonate the user, the factors are generally chosen from different categories in the following list:

- *Something the user knows*: This is generally a password.
- *Something the user has:* This can be a security token, a specific device, a swipe card, or a security key.
- *Something inherent to the user:* This is typically addressed using a physical characteristic, such as a fingerprint or iris scan, or facial renunciation.

There is wide scope for each category. An inherent characteristic could be the user's location, for example, although this can be difficult to establish.

The most common combination is something the user knows and something the user has, and this is generally implemented as a password and a specific device. In this chapter, I show you how to use confirmation codes as the second factor, which relies on the user having a specific phone number to receive the codes via SMS. In Chapter 21, I show you how to use an authenticator application, which is a more sophisticated alternative.

MULTIFACTOR VS. TWO-FACTOR AUTHENTICATION

Recently, the term *two-factor* has been replaced by *multifactor*, which is intended to indicate that some situations may require more than two factors to provide an acceptable level of protection.

This is well-intentioned, but each additional requirement you add makes it more difficult for legitimate users to sign in and adds to the application's support and administration overhead.

My advice is to start with two factors and get the behaviors established with your users before attempting to ratchet up the security with additional factors. And, as I have already noted, the harder you make it to use an application, the more effort your users will make to subvert your security policies. There are no benefits to three or more factors if you don't have the willing cooperation of your users.

Updating the User Class

Identity allows two-factor authentication to be configured for individual users. To keep track of two-factor authentication, add the property shown in Listing 20-13 to the AppUser class.

Listing 20-13. 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; }

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

        public bool TwoFactorEnabled { get; set; }
    }
}
```

In Listing 20-14, I have updated the seed data added to the store to set the two-factor property for the example users.

Listing 20-14. Updating the Seed Data in the UserStore.cs File in the Identity/Store Folder

```
...
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" };
    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$");
        users.TryAdd(user.Id, user);
    }
}
...
```

The changes enable two-factor authentication for Alice and Charlie.

Extending the User Store to Support Two-Factor Authentication

The next step is to extend the user store to implement the IUserTwoFactorStore<T> interface, which is used to keep track of which users are required to use two-factor authentication and which defines the methods described in Table 20-8. As with all the other user store interfaces I have described, these methods define a CancellationToken parameter named token that is used to receive a notification when an asynchronous task is canceled.

Table 20-8. *The IUserTwoFactorStore<T> Methods*

Name	Description
`GetTwoFactorEnabledAsync(user, token)`	This method returns `true` if the specified user should use two-factor authentication.
`SetTwoFactorEnabledAsync(user, enabled, token)`	This method sets the two-factor authentication requirement for the specified user.

Add a class file named `UserStoreTwoFactor.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 20-15.

Listing 20-15. The Contents of the UserStoreTwoFactor.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 : IUserTwoFactorStore<AppUser> {

        public Task<bool> GetTwoFactorEnabledAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.TwoFactorEnabled);

        public Task SetTwoFactorEnabledAsync(AppUser user, bool enabled,
                CancellationToken token) {
            user.TwoFactorEnabled = enabled;
            return Task.CompletedTask;
        }
    }
}
```

The implementation of the interface maps the methods described in Table 20-8 into the property added to the `AppUser` class in the previous section.

USING SMS FOR TWO-FACTOR AUTHENTICATION

There are differing opinions about the use of SMS for two-factor authentication. Without a doubt, SMS is susceptible to a range of attacks, and there have been documented examples of phones being cloned or transferred so that confirmation codes can be intercepted.

On the other hand, SMS is cheap, effective, and almost universally available. Smartphone authenticators (described in Chapter 21) or hardware security tokens may be more secure, but not all users have smartphones, and not all projects can supply users with tokens.

There were headlines in 2016 when a draft document from the National Institute of Standards and Technology (NIST) deprecated the use of SMS for two-factor authentication, and this has led some to believe that SMS should no longer be used. However, in the final release of that document, SMS was not deprecated, but attention was drawn to the risks inherent in SMS, such as SIM changes and phone number porting. (See `https://pages.nist.gov/800-63-3` for details.)

You should perform risk assessments for your projects, but my advice is not to dismiss SMS unless you have a well-understood security requirement and have tight control over the devices your users carry and the apps they can install.

Managing Two-Factor Authentication

The `UserManager<T>` method provides a set of members for managing the two-factor authentication settings in the user store, as described in Table 20-9.

Table 20-9. *The UserManager<T> members for Managing Two-Factor Authentication*

Name	Description
`SupportsUserTwoFactor`	This property returns true if the user store implements the `IUserTwoFactorStore<T>` interface, where `T` is the user class.
`GetTwoFactorEnabledAsync(user)`	This method calls the user store's `GetTwoFactorEnabledAsync` method to get the two-factor setting for the specified user.
`SetTwoFactorEnabledAsync(user, enabled)`	This method sets the two-factor setting for the specified user by calling the user store's `SetTwoFactorEnabledAsync` method, after which the security stamp is updated, and the user manager's update sequence is performed.
`GenerateTwoFactorTokenAsync(user, provider)`	This method generates a security token that the user will provide to identify themselves in the final step of the process described in the next section.

The `UserManager<T>` class provides only basic features because the important two-factor features are handled by the `SignInManager<T>` class, which I describe shortly. To manage the two-factor authentication settings, add a Razor View named `_EditUserTwoFactor.cshtml` to the `ExampleApp/Pages/Store` folder with the content shown in Listing 20-16.

Listing 20-16. The Contents of 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>
}
```

Following the pattern used throughout this part of the book, I am working directly with the properties defined by the `AppUser` class and not the methods described in Table 20-9, although I use the `SupportsUserTwoFactor` property to confirm that the user store supports two-factor settings. Add the element shown in Listing 20-17 to the `EditUser.cshtml` file to incorporate the new view into the application.

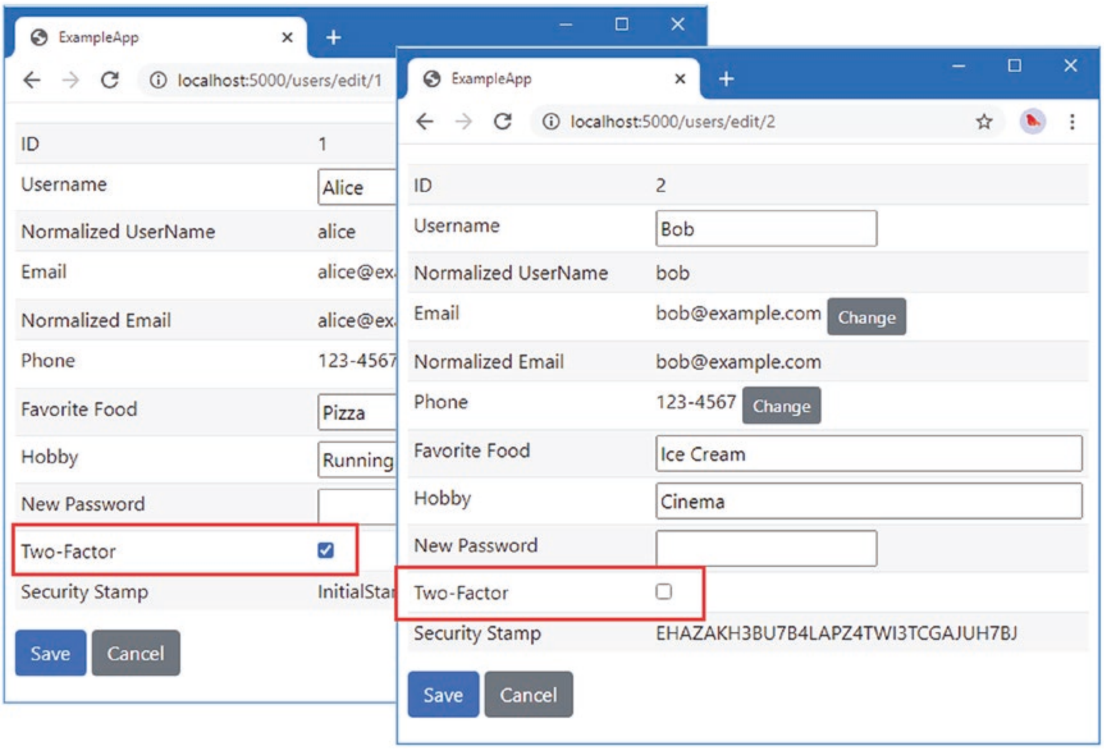
Listing 20-17. Adding a Partial View in the EditUser.cshtml File in the Pages/Store Folder

```
@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="_EditUserTwoFactor" 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>
```

Restart ASP.NET Core, request `http://localhost:5000/users`, and click the Edit button for one of the users. You will see a checkbox that allows the two-factor authentication option to be changed, as shown in Figure 20-5.



ExampleApp

localhost:5000/users/edit/1

ID	1
Username	Alice
Normalized UserName	alice
Email	alice@ex
Normalized Email	alice@ex
Phone	123-4567
Favorite Food	Pizza
Hobby	Running
New Password	
Two-Factor	☑
Security Stamp	InitialSta

ExampleApp

localhost:5000/users/edit/2

ID	2
Username	Bob
Normalized UserName	bob
Email	bob@example.com Change
Normalized Email	bob@example.com
Phone	123-4567 Change
Favorite Food	Ice Cream
Hobby	Cinema
New Password	
Two-Factor	☐
Security Stamp	EHAZAKH3BU7B4LAPZ4TWI3TCGAJUH7BJ

Figure 20-5. *Managing the two-factor authentication setting*

Signing In with Two Factors

Now that the foundation is in place, it is time to change the sign-in process to support two-factor authentication for those users who require it. It is helpful to focus on the process before diving into the way it is implemented. There are three steps.

1. The first step is a normal password sign-in. The user provides their account name and their password to the application, which are validated against the data in the user store. The user is signed into the application if they don't require two-factor authentication, and the process ends. The process also ends if the user has provided the wrong password. If the user has provided the correct password and requires two-factor authentication, a cookie identifying the user is added to the response, and the client is sent a redirection to the URL for the next step of the process.

2. The request the client sends to the next URL contains the cookie, which is extracted from the request and used to identify the user. This cookie is only used for two-factor sign-ins and doesn't allow the user to sign in to the application yet. A confirmation code is sent to the user using the contact information in the user store associated with the user identified by the cookie created in the previous step. The client is sent a redirection to the URL for the next step in the process.

3. The client is prompted for the security code they have been sent, and the user account is once again identified by the cookie that was created in the first step. If the code is valid, the user is signed into the application. A cookie is added to the response that will authenticate subsequent requests and grant the user access to the application, completing the sign-in process. If the code is invalid, the application can elect to prompt the user again or terminate the process.

The part of the process that causes confusion is the cookie, which is used to identify the user but doesn't grant them access to the application. This cookie is created using the standard Identity authentication features but uses a different cookie name and contains only enough (encrypted) data to identify the user without including any additional information about the user account and its roles and claims. Table 20-10 describes the methods that the `SignInManager<T>` class provides for the two-factor sign in process.

Table 20-10. *The SignInManager<T> Methods for Two-Factor Sign-In*

Name	Description
`PasswordSignInAsync(user, password, persist, lockout)`	This is the standard password sign-in method, which returns a `SignInResult` object that indicates whether two-factor authentication is required and adds the cookie to the response.
`GetTwoFactorAuthenticationUserAsync()`	This method retrieves user information from the cookie created by the first phase of the two-factor sign-in process.
`TwoFactorSignInAsync(provider, code persistent, remember)`	This method validates a code for the specified user and signs the user into the application if it is valid. The `persistent` argument determines if the authentication cookie will persist when the browser is closed, and the `remember` argument determines whether a cookie is created that will bypass the sign-in process, as described in the next section.

ASP.NET Core Identity supports an optional feature that adds a cookie to the response once a user has completed two-factor authentication and that bypasses the two-factor requirement the next time the user signs in from the same client. When this feature is used, the user is signed into the application with just a regular password sign-in.

This is a useful feature for making two-factor signing in more palatable to users because they only have to provide the second factor when the cookie expires or if they sign in from a different client. Creating the cookie to remember clients is controlled by the `remember` argument passed to the `TwoFactorSignInAsync` method described in Table 20-10 and can also be managed through the `SignInManager<T>` methods described in Table 20-11.

Table 20-11. *The SignInManager<T> Methods for Remembering Two-Factor Clients*

Name	Description
`RememberTwoFactorClientAsync(user)`	This method adds a cookie to the response to indicate that the user has completed a two-factor sign-in. This method is called automatically when the `remember` argument to the `TwoFactorSignInAsync` method is `true`.
`IsTwoFactorClientRememberedAsync(user)`	This method inspects the request to see if it contains a cookie that indicates the current user has previously completed a two-factor sign in and should be allowed to bypass the second factor.
`ForgetTwoFactorClientAsync(user)`	This method deletes the cookie that remembers the client.

Creating the Two-Factor Token Generator

The first step is to create the generator for the tokens that will be given to the user in step 2 of the sign-in process, which is done with an implementation of the `IUserTwoFactorTokenProvider<T>` interface. For this chapter, I am going to build on the `SimpleTokenGenerator` class. Add a class file named `TwoFactorSignInTokenGenerator.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 20-18.

Listing 20-18. The Contents of the TwoFactorSignInTokenGenerator.cs File in the Identity Folder

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

namespace ExampleApp.Identity {

    public class TwoFactorSignInTokenGenerator : SimpleTokenGenerator {

        protected override int CodeLength => 3;

        public override Task<bool> CanGenerateTwoFactorTokenAsync(
                UserManager<AppUser> manager, AppUser user) {
            return Task.FromResult(user.TwoFactorEnabled);
        }
    }
}
```

The key difference from the token generators from earlier examples is the implementation of the `CanGenerateTwoFactorTokenAsync` method, which tells Identity that tokens are available for any user whose `TwoFactorEnabled` property is `true`.

Configuring the Token Generator and Cookie Authentication Handler

The ASP.NET Core cookie authentication handler can be used to create the cookies required by the two-factor sign-in process. Listing 20-19 shows the configuration changes required in the `ConfigureServices` method of the `Startup` class to set up the sign-in cookies and register the two-factor token generator.

Listing 20-19. Configuring the Application 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>();

    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)
    .AddSignInManager()
    .AddRoles<AppRole>();

    services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();
    services.AddSingleton<IPasswordValidator<AppUser>, PasswordValidator>();
    services.AddScoped<IUserClaimsPrincipalFactory<AppUser>,
        AppUserClaimsPrincipalFactory>();
    services.AddSingleton<IRoleValidator<AppRole>, RoleValidator>();

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

The IdentityConstants class defines two properties that are used as authentication scheme names. The TwoFactorUserIdScheme property identifies the scheme used for the cookie sent to the client after a successful password login in stage 1 of the two-factor process. The TwoFactorRememberMeScheme property specifies the scheme used for the cookie that remembers clients after a successful two-factor login. Authentication handlers are required for schemes, even if the application doesn't remember clients.

Changing the Password Sign-In Process

To support two-factor sign-ins, I need to check the SignInResult object produced by the SignInManager<T>.PasswordSignInAsync to see if the RequiresTwoFactor property is true. If it is, then I need to redirect the browser to the URL that handles the second factor. Listing 20-20 shows the changes to the POST handler method for the SignIn Razor Page.

Listing 20-20. Supporting Two-Factors in the SignIn.cshtml.cs File in the Pages Folder

```
...
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) {
        if (result.IsLockedOut) {
            TimeSpan remaining = (await UserManager
                .GetLockoutEndDateAsync(user))
                .GetValueOrDefault().Subtract(DateTimeOffset.Now);
            Message = $"Locked Out for {remaining.Minutes} mins and"
                + $" {remaining.Seconds} secs";
        } else if (result.RequiresTwoFactor) {
            return RedirectToPage("/SignInTwoFactor", new { returnUrl = returnUrl });
        } else if (result.IsNotAllowed) {
            Message = "Sign In Not Allowed";
        } else {
            Message = "Access Denied";
        }
        return Page();
    }
    return Redirect(returnUrl ?? "/signin");
}
...
```

It is important to perform the redirection only when the `RequiresTwoFactor` property is `true` and not to do so automatically when the user has the two-factor requirement enabled. This is because a failed password sign doesn't progress to the next stage and because the feature that remembers successful two-factor sign-ins may allow the user to sign in with just a password.

Supporting the Second Factor

Add a Razor Page named `SignInTwoFactor.cshtml` to the `ExampleApp/Pages` folder with the content shown in Listing 20-21.

Listing 20-21. The Contents of 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>

<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="smscode"/>
        </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>
```

The user is presented with an `input` element that allows a code to be entered, and buttons to sign in, cancel, or send a new code. To define the page model for the Razor Page, add the code shown in Listing 20-22 to the `SignInTwoFactor.cshtml.cs` file in the `Pages` folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 20-22. The Contents of 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 async Task OnGet() {
            AppUser user = await SignInManager.GetTwoFactorAuthenticationUserAsync();
            if (user != null) {
                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 smscode, string rememberMe,
                [FromQuery] string returnUrl) {
            AppUser user = await SignInManager.GetTwoFactorAuthenticationUserAsync();
            if (user != null) {
                SignInResult result = await SignInManager.TwoFactorSignInAsync(
                    IdentityConstants.TwoFactorUserIdScheme, smscode, true,
                        !string.IsNullOrEmpty(rememberMe));
                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();
        }
    }
}
```

The GET handler method uses the GetTwoFactorAuthenticationUserAsync method to retrieve details of the user who is signing in. It is important to remember that the user isn't signed into the application until they have completed the entire process and that only this method should be used to get the user's details. Once the user details have been obtained, the user's security token is updated to invalidate any previous tokens, and the GenerateTwoFactorTokenAsync method is used to generate a token that is sent to the user through the ISMSSender service, which simulates sending SMS messages by writing console messages.

The POST handler method receives the security code, the user's choice for remembering the client, and, optionally, the URL to return to once sign-in is complete. The GetTwoFactorAuthenticationUserAsync method is used to get the user's details again, and the TwoFactorSignInAsync method is used to validate the token and complete the sign-in process.

Forgetting the Client

The final step is to give the user the option to forget the client when they log out, ensuring that the full two-factor sign-in process is required when they next log in. Add the elements shown in Listing 20-23 to the SignOut Razor Page.

Listing 20-23. Adding Elements in the SignOut.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignOutModel

<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-check m-2">
            <input class="form-check-input" type="checkbox" name="forgetMe" />
            <label class="form-check-label">Forget Me</label>
        </div>
        <button class="btn btn-info" type="submit">Sign Out</button>
    </form>
</div>
```

The changes in Listing 20-24 update the page model class so that the POST handler method receives the value from the checkbox and forgets the client.

Listing 20-24. Forgetting the Client in the SignOut.cshtml.cs File in the Pages Folder

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

namespace ExampleApp.Pages {
    public class SignOutModel : PageModel {
        public string Username { get; set; }
        public SignOutModel(SignInManager<AppUser> manager)
            => SignInManager = manager;

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

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

        public async Task<ActionResult> OnPost(string forgetMe) {
            if (!string.IsNullOrEmpty(forgetMe)) {
                await SignInManager.ForgetTwoFactorClientAsync();
            }
            await HttpContext.SignOutAsync();
            return RedirectToPage("SignIn");
        }
    }
}
```

The `ForgetTwoFactorClientAsync` method is used to delete the cookie that allows the user to bypass part of the two-factor process.

Testing Two-Factor Sign-In

Restart ASP.NET Core, request `http://localhost:5000/signout`, and click the Sign Out button to delete any existing cookies created by earlier examples. Next, request `http://localhost:5000/secret`, which will produce a challenge response and begin the two-factor process. Select alice@example.com with the `select` element, enter MySecret1$ into the password field, and click the Sign In button. The password will be validated, and you will be prompted to enter the security code. Look at the console output, and you will see a message like this:

```
--- SMS Starts ---
To: 123-4567
Your security code is A06E9C
--- SMS Ends ---
```

Enter the security code shown in your message, which will be different from the one shown here, select the Remember Me option, and click the Sign In button. The sign-in process is complete, and you will be redirected to the `/secret` URL, as shown in Figure 20-6.

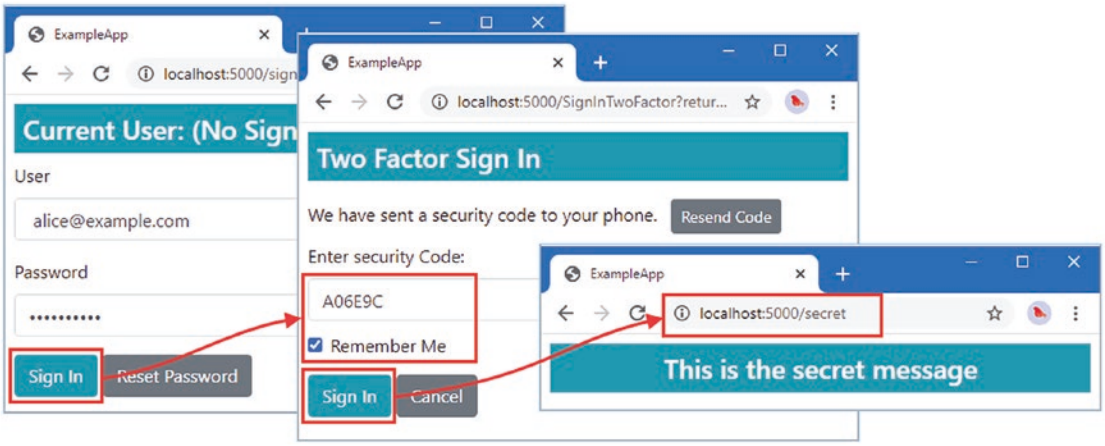


Figure 20-6. *The two-factor sign-in process*

To test the feature that remembers the client, request `http://localhost:5000/signout`, and click the Sign Out button without enabling the checkbox. Request `http://localhost:5000/secret`, select alice@example.com when presented with the challenge response, and enter `MySecret1$` into the password field. Click Sign In, and you will be redirected to the `/secret` URL without being prompted for a security code, as shown in Figure 20-7.

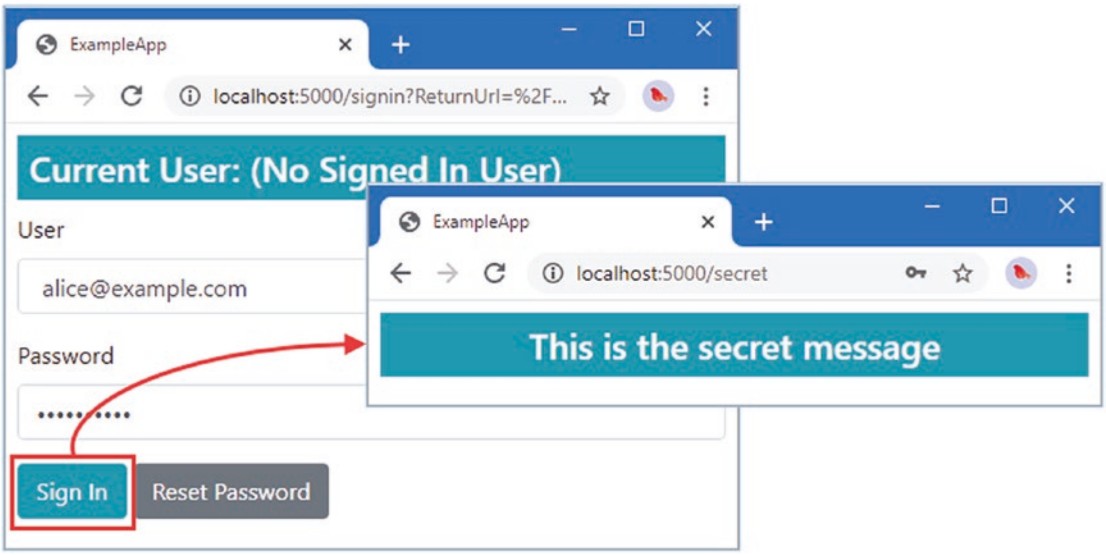

ExampleApp
localhost:5000/signin?ReturnUrl=%2F...
Current User: (No Signed In User)
User
alice@example.com
Password
Sign In
Reset Password
localhost:5000/secret
This is the secret message

Figure 20-7. *The sign-in process with a remembered client*

If you check the option to forget the client, you will be prompted for the security code again.

Restricting the Scope of Remembered Clients

Remembering clients is a useful feature for making signing in easier for users, but you may want to force a complete two-factor sign in for especially sensitive operations. The `SignInManager<T>` class adds a claim to the `ClaimsPrincipal` object that represents the user, making it possible to determine whether a remembered client allowed the user to skip part of the two-factor process. In Listing 20-25, I have defined an authorization policy that checks for the claim added by the `SignInManager<T>` class when the full two-factor sign-in process has been performed.

Listing 20-25. Creating an Authorization Policy in the Startup.cs File in the ExampleApp Folder

```
...
services.AddAuthorization(opts => {
    AuthorizationPolicies.AddPolicies(opts);
    opts.AddPolicy("Full2FARequired", builder => {
        builder.RequireClaim("amr", "mfa");
    });
});
...
```

The policy requires a claim whose type is `amr` with the value `mfa`. The term AMR refers to Authentication Method Reference, which is specified by RFC8176, which standardizes claims that describe authentication methods. (See `https://tools.ietf.org/html/rfc8176` for details.) The `mfa` value indicates the user has signed in using multifactor authentication. The user will still have an `amr` claim even if they have bypassed the second factor or are configured for single-factor sign-ins, but the claim value will be `pwd`, indicating password-based authentication.

Defining an Authorization Failed Landing Page

The standard authorization features cannot be used in this situation because the user is signed in as far as Identity is concerned, which means that failing to meet the authorization requirement defined in Listing 20-25 will result in a forbidden response, instead of a challenge response that will allow the user to go through the full two-factor process. Instead, I need a landing page that will explain to the user that a full login is required and sign them out so they can sign in again without the client being remembered. Add a Razor Page named `Full2FARequired.cshtml` to the `ExampleApp/Pages` folder with the content shown in Listing 20-26.

Listing 20-26. The Contents of the Full2FARequired.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.Full2FARequiredModel

<h4 class="bg-primary text-white text-center p-2">Two-Factor Sign In Required</h4>

<form method="post">
    <div class="m-2">
        The full two-factor sign in process is required. Click the
        OK button to sign out of the application so you can sign in
        using with a password and a security code.
    </div>
    <button class="btn btn-primary mx-2" type="submit">OK</button>
</form>
```

The page displays an explanatory message and a button the user will click to sign out of the application and forget the client. To define the page model class, add the code shown in Listing 20-27 to the `Full2FARequired.cshtml.cs` file in the `Pages` folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 20-27. The Contents of the Full2FARequired.cshtml.cs File in the Pages Folder

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

namespace ExampleApp.Pages {

    public class Full2FARequiredModel : PageModel {

        public Full2FARequiredModel(UserManager<AppUser> userManager,
                SignInManager<AppUser> signInManager) {
            UserManager = userManager;
            SignInManager = signInManager;
        }

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

        public async Task<IActionResult> OnPostAsync(string returnUrl) {
            AppUser user = await UserManager.GetUserAsync(HttpContext.User);
            if (await SignInManager.IsTwoFactorClientRememberedAsync(user)) {
                await SignInManager.ForgetTwoFactorClientAsync();
            }
            await HttpContext.SignOutAsync();
            return Redirect($"/signin?returnUrl={returnUrl}");
        }
    }
}
```

The POST handler method uses the methods provided by the `SignInManager<T>` class to forget the client and then signs the user out through the `HttpContext.SignOutAsync` method.

■ **Caution** Do not use the `SignInManager<T>.SignOutAsync` method to sign out of the application because it will throw an exception, reporting there is no handler for the external scheme.

Creating the Page and Action Filters

The standard authorization features are not useful in this situation, and I need to create a page filter to apply the policy in Razor Pages and an action filter to apply the policy in controllers. There is enough common code in page and action filters to allow a single class to be used for both. Add a class file named `Full2FARequiredFilterAttribute.cs` to the `Examples/Identity` folder and use it to define the code shown in Listing 20-28.

Listing 20-28. The Contents of the Full2FARequiredFilterAttribute.cs File in the Identity Folder

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;

namespace ExampleApp.Identity {

    public class Full2FARequiredFilterAttribute : Attribute,
            IAsyncPageFilter, IAsyncActionFilter {

        public async Task OnActionExecutionAsync(ActionExecutingContext context,
                ActionExecutionDelegate next) {
            IActionResult result = await ApplyPolicy(context.HttpContext);
            if (result != null) {
                context.Result = result;
            } else {
                await next.Invoke();
            }
        }

        public async Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext
                context, PageHandlerExecutionDelegate next) {
            IActionResult result = await ApplyPolicy(context.HttpContext);
            if (result != null) {
                context.Result = result;
            } else {
                await next.Invoke();
            }
        }

        public async Task<IActionResult> ApplyPolicy(HttpContext context) {
            IAuthorizationService authService =
                context.RequestServices.GetService<IAuthorizationService>();
            if (!(await authService.AuthorizeAsync(context.User,
                 "Full2FARequired")).Succeeded) {
                return new RedirectToPageResult("/Full2FARequired",
                    new { returnUrl = Path(context) });
            }
            return null;
        }

        public Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context) {
            return Task.CompletedTask;
        }
        private string Path(HttpContext context) =>
            $"{context.Request.Path}{context.Request.QueryString}";
    }
}
```

The filter uses the `IAuthorizationService` service to enforce the `Full2FARequired` policy and creates a redirection to the landing page if the policy requirements are not met.

Applying the Filter to a Page

To create a resource that will require a full two-factor sign-in, add a Razor Page named `VerySecret.cshtml` in the `ExampleApp/Pages` folder with the content shown in Listing 20-29.

Listing 20-29. The Contents of the VerySecret.cshtml File in the Pages Folder

```
@page
@using Microsoft.AspNetCore.Authorization
@using ExampleApp.Identity
@attribute [Authorize]
@attribute [Full2FARequiredFilter]

<h4 class="bg-info text-center text-white m-2 p-2">
    This is the VERY secret message
</h4>
```

The application of the `Authorize` attribute ensures that users that are not signed in are presented with the standard challenge response. The `Full2FARequiredFilter` deals with two-factor users who bypassed the second factor because their client was remembered by the application.

Testing the Full Two-Factor Requirement

The process for testing the full two-factor requirement is complex and requires multiple steps, each of which must be followed exactly. The goal is to have logged in with two factors and have the application remember the client, sign out, log back in without needing to provide the second factor, and then request the `VerySecret` page. Here are the steps required to perform the test:

1. Restart ASP.NET Core.
2. Request `http://localhost:5000/signout`, select the Forget Me option, and click the Sign Out button.
3. Request `http://localhost:5000/verysecret`. You will receive the standard challenge response.
4. Select alice@example.com using the element, enter MySecret1$ into the password field, and click the Sign In button.
5. Enter the security code that is written to the console into the Enter Security Code text field displayed by the browser. Check the Remember Me option and click the Sign In button.

   This is not the end of the test process, but, at this point, you are authenticated using both factors and will see the content produced by the `VerySecret` Razor Page, as shown in Figure 20-8.

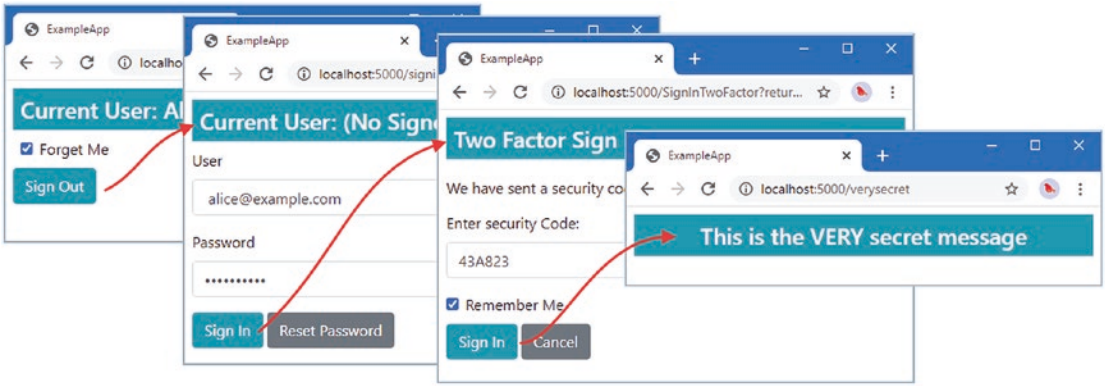
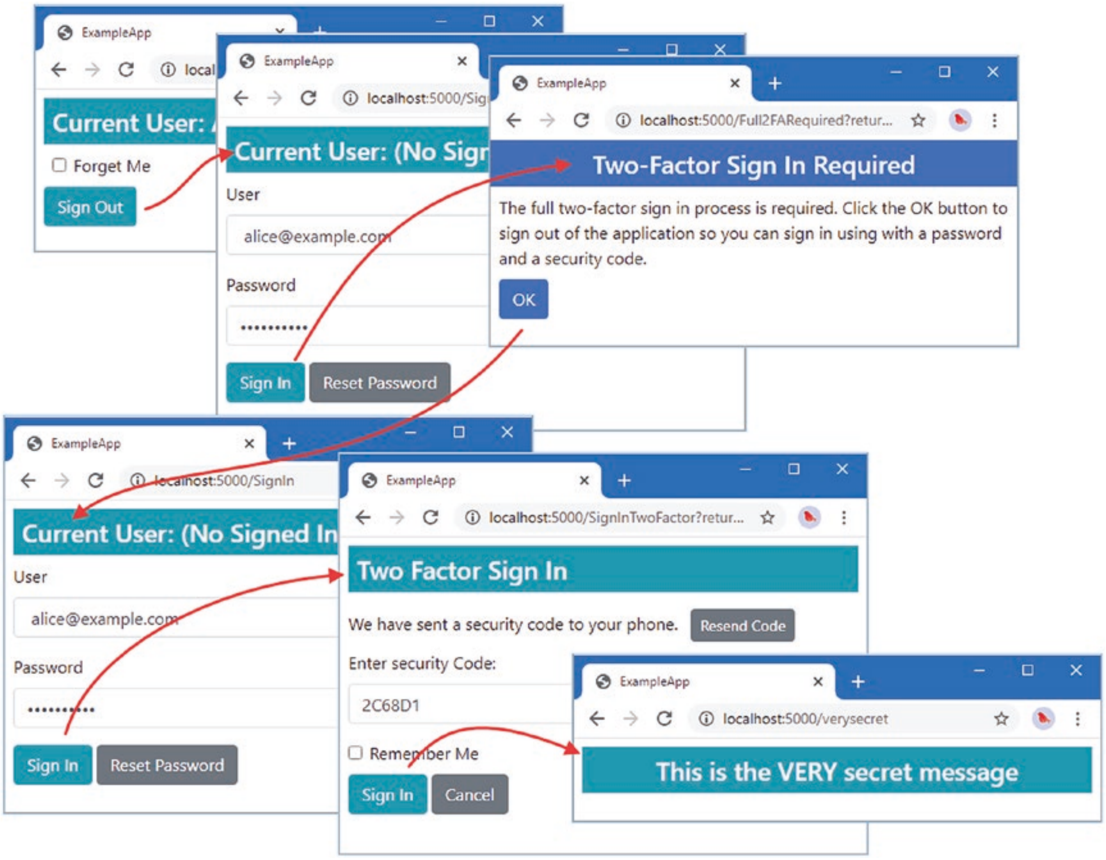


Figure 20-8. *The first part of the test process*

6. Request `http://localhost:5000/signout`, ensure the Forget Me option is unchecked, and click the Sign Out button. You will be redirected to the `SignIn` Razor Page.

7. Select alice@example.com using the element, enter MySecret1$ into the password field, and click the Sign In button. You will be authenticated without providing a second factor because your client has been remembered.

8. Request `http://localhost:5000/verysecret`. Instead of seeing the requested page, you will be redirected to the landing page. Click OK, go through the two-factor sign in process, and, once authenticated, you will see the protected content, as shown in Figure 20-9.


Two-Factor Sign In Required

The full two-factor sign in process is required. Click the OK button to sign out of the application so you can sign in using with a password and a security code.

Two Factor Sign In

We have sent a security code to your phone.

Enter security Code: 2C68D1

This is the VERY secret message
ExampleApp
localhost:5000/Full2FARequired?retur...
localhost:5000/SignIn
localhost:5000/SignInTwoFactor?retur...
localhost:5000/verysecret
Current User: (No Signed In
Forget Me
Sign Out
User
alice@example.com
Password
Sign In
Reset Password
OK
Resend Code
Remember Me
Cancel

Figure 20-9. *The second part of the test process*

The test process is complex, but the overall result is that the `VerySecret` Razor Page can be accessed only by users who have signed in by providing two factors.

Summary

In this chapter, I explained how lockouts are implemented and showed you how to restrict access to accounts that have been confirmed. I also demonstrated the features that Identity uses to support two-factor authentication, for which I generated simulated SMS messages. In the next chapter, I continue on the theme of two-factor authentication and describe how authenticators and recovery codes are implemented.