CHAPTER 17

Claims, Roles, and Confirmations

In this chapter, I continue to add features to the store and build administration tools that use those features through the `UserManager<T>` class. I start by adding support for storing and managing claims and then use these features to add support for roles, which are given special status in ASP.NET Core and ASP.NET Core Identity because they are so widely used. I also show you how to generate and validate confirmation tokens, which are used to ensure that a user can access the email address or phone number they provide. Table 17-1 puts these features in context.

Table 17-1. *Putting Claims, Roles, and Confirmations in Context*

Question	Answer
What are they?	Roles restrict access to protected resources. Claims store additional data about a user, including roles. Confirmations are tokens sent to the user, which they present back to the application to prove their identity.
Why are they useful?	Roles are widely used to define authorization policies. Claims can also be used for authorization and can be used to store any data about the user. Confirmations ensure the user can be reached through the email address or phone number in the user store.
How are they used?	For roles and claims, the store implements optional interfaces. Confirmations require custom workflows that generate and validate change tokens.
Are there any pitfalls or limitations?	Care must be taken to deal with the normalization of role names, as explained in the "Understanding the Role Normalization Pitfall" section.
Are there any alternatives?	These are optional features and are not required if you do not require role-based access control, need to store additional data, or need to check the user's email address and phone number.

Table 17-2 summarizes the chapter.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_17

Table 17-2. *Chapter Summary*

Problem	Solution	Listing
Support claims in the user store	Implement the `IUserClaimStore<T>` interface.	2–8
Support roles in the user store	Implement the `IUserRoleStore<T>` interface.	9–12
Avoid role name normalization issues	Store roles as claims.	13
Support confirmations in the store	Implement the `IUserSecurityStampStore<T>`, create token generators, and send the tokens the user using the contact details they provide.	14–30

Preparing for This Chapter

This chapter uses the `ExampleApp` project from Chapter 16. 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 17-1 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 17-1. 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 17-1. The data is stored only in memory, and changes will be lost when ASP.NET Core is stopped.

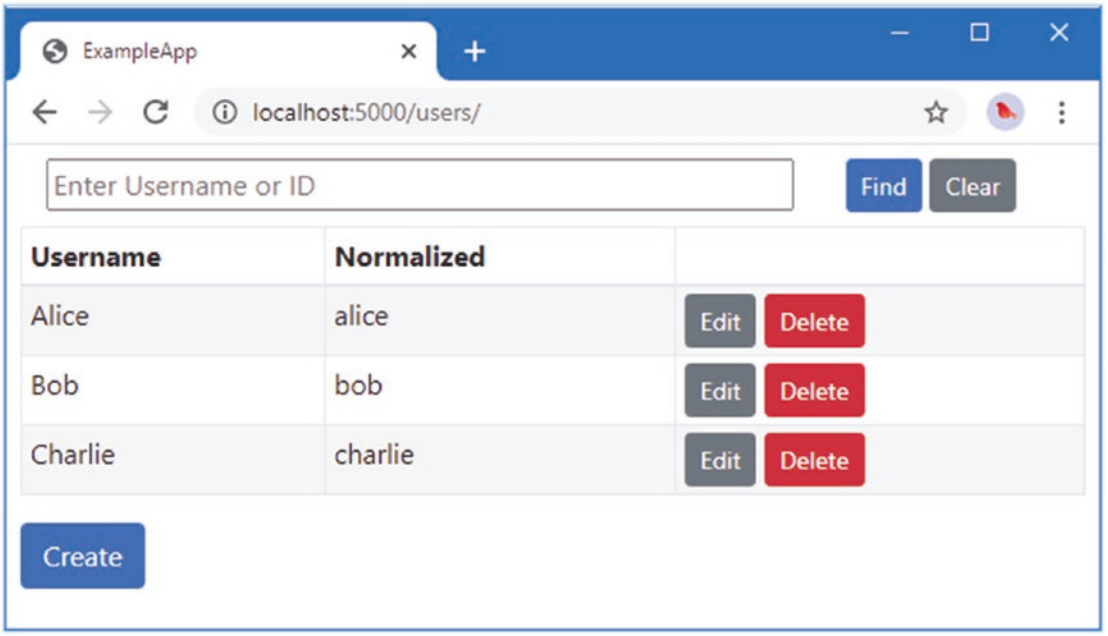

Figure 17-1. *Running the example application*

Storing Claims in the User Store

Now that I have the basic features in place, I can start to add features to the store, starting with support for storing a user's claims. Stores that can manage claims implement the `IUserClaimStore<T>` interface, where `<T>` is the user class. Table 17-3 describes the methods defined by the `IUserClaimStore<T>` interface. As with earlier store interfaces, these asynchronous methods define a `CancellationToken` parameter that is used to receive notifications when an operation is cancelled and is given the name `token` in the table.

Table 17-3. *The IUserClaimStore<T> Methods*

Name	Description
`GetClaimsAsync(user, token)`	This method returns an `IList<Claim>` that contains all the user's claims.
`AddClaimsAsync(user, claims, token)`	This method adds one or more claims to the user object, which is received as an `IEnumerable<Claim>`.
`RemoveClaimsAsync(user, claims, token)`	This method removes one or more claims to the user object, which is received as an `IEnumerable<Claim>`.
`ReplaceClaimAsync(user, oldClaim, newClaim, token)`	This method replaces one claim with another. The claims do not have to be of the same type.
`GetUsersForClaimAsync(claim, token)`	This method returns an `IList<T>` object, where `T` is the user class that contains all the users with a specific claim.

To prepare for storing claims, Listing 17-2 extends the user class to support claims.

Listing 17-2. Adding Claims 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; }
    }
}
```

To extend the example user store to support claims, add a class file named `UserStoreClaims.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 17-3.

Listing 17-3. The Contents of the UserStoreClaims.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {
    public partial class UserStore : IUserClaimStore<AppUser>,
            IEqualityComparer<Claim> {

        public Task AddClaimsAsync(AppUser user, IEnumerable<Claim> claims,
                CancellationToken token) {
            if (user.Claims == null) {
                user.Claims = new List<Claim>();
            }
```

```
            foreach (Claim claim in claims) {
                user.Claims.Add(claim);
            }
            return Task.CompletedTask;
        }

        public Task<IList<Claim>> GetClaimsAsync(AppUser user,
                CancellationToken token) => Task.FromResult(user.Claims);

        public Task RemoveClaimsAsync(AppUser user, IEnumerable<Claim> claims,
                CancellationToken token) {
            foreach (Claim c in user.Claims.Intersect(claims, this).ToList()) {
                user.Claims.Remove(c);
            }
            return Task.CompletedTask;
        }

        public async Task ReplaceClaimAsync(AppUser user, Claim oldclaim,
                Claim newClaim, CancellationToken token) {
            await RemoveClaimsAsync(user, new[] { oldclaim }, token);
            user.Claims.Add(newClaim);
        }

        public Task<IList<AppUser>> GetUsersForClaimAsync(Claim claim,
                CancellationToken token) =>
            Task.FromResult(
                Users.Where(u => u.Claims.Any(c => Equals(c, claim)))
                   .ToList() as IList<AppUser>);

        public bool Equals(Claim first, Claim second) =>
            first.Type == second.Type && string.Equals(first.Value, second.Value,
                    StringComparison.OrdinalIgnoreCase);

        public int GetHashCode(Claim claim) =>
            claim.Type.GetHashCode() + claim.Value.GetHashCode();
    }
}
```

Identity doesn't enforce restrictions on how a store manages claims, but it is important to understand that the same claim can appear more than once. In Listing 17-3, I have used a dictionary to keep track of objects, using the `AppUser.Id` property as the key to store collections of `Claim` objects.

To seed the store with claims, add the statements shown in Listing 17-4 to the `UserStore.cs` file, which contains the constructor and seed data for the user store class.

Listing 17-4. Seeding Claims 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 UserStore(ILookupNormalizer normalizer) {
            Normalizer = normalizer;
            SeedStore();
        }

        private void SeedStore() {

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

            int idCounter = 0;

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

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

The new code populates the store with `Role` claims using seed data defined in the `UsersAndClaims` class.

Managing Claims in the User Store

The UserManager<T> class provides a set of methods that provide access to claims when the user store class implements the IUserClaimStore<T> interface, as described in Table 17-4, along with a property that allows support for claims to be checked.

Table 17-4. *The UserManager<T> Members for Claims*

Name	Description
SupportsUserClaim	This property returns true if the user store implements the IUserClaimStore<T> interface.
GetClaimsAsync(user)	This method returns the list of claims for the specified user by calling the store's GetClaimsAsync method.
GetUsersForClaimAsync(claim)	This method returns the list of users that have the specified claim by calling the store's GetUsersForClaimAsync method.
AddClaimsAsync(user, claims)	This method adds multiple claims to the specified user and commits the change to the store by calling the store's AddClaimsAsync method, after which the user manager's update sequence is performed.
AddClaimAsync(user, claim)	This convenience method creates an array containing the single claim and passes it to the AddClaimsAsync method.
ReplaceClaimAsync(user, oldClaim, newClaim)	This method replaces a claim for the specified user by calling the store's ReplaceClaimAsync method, after which the user manager's update sequence is performed.
RemoveClaimsAsync(user, claims)	This method removes multiple claims from the specified user and commits the change to the store by calling the store's RemoveClaimsAsync method, after which the user manager's update sequence is performed.
RemoveClaimAsync(user, claim)	This convenience method creates an array containing the claim and calls the RemoveClaimsAsync method.

To manage claims, add a Razor View named _ClaimsRow.cshtml to the Pages/Store folder and use it to define the partial view shown in Listing 17-5. If you are using Visual Studio, you can create this file using the Razor View – Empty item template.

Listing 17-5. The Contents of 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>
    </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>
```

This partial view produces a set of table cells that contain a `select` element and a text field for a claim. The `select` element allows the type of a claim to be selected, and the text field allows the value to be edited. There are buttons in each row to save changes or delete the claim. Claims can have any string value as their type, but to keep the example simple, the partial view only supports the `Role`, `GivenName`, and `Surname` claim types, using values from the `ClaimTypes` class.

Add a Razor Page named `Claims.cshtml` to the `Pages/Store` folder with the contents shown in Listing 17-6.

Listing 17-6. The Contents of the Claims.cshtml File in the Pages/Store Folder

```
@page "/users/claims/{id?}"
@model ExampleApp.Pages.Store.ClaimsModel

@{
    Claim newClaim = new Claim(string.Empty, string.Empty);
}

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

<div class="m-2">
    <table class="table table-sm table-striped">
        <thead><tr><th>Type</th><th>Value</th><th/></tr></thead>
        <tbody>
            @foreach (Claim claim in Model.Claims) {
                <tr>
                    <partial name="_ClaimsRow"
                        model="@((Model.AppUserObject.Id, claim, false))" />
                </tr>
            }
```

```
            <tr>
                <partial name="_ClaimsRow"
                    model="@((Model.AppUserObject.Id, newClaim, true))" />
            </tr>
        </tbody>
    </table>
    <div>
        <a asp-page="users" class="btn btn-secondary">Back</a>
    </div>
</div>
```

The Razor Page uses the `_ClaimsRow` partial view to display a user's claims. To define the page model, add the code shown in Listing 17-7 to the `Claims.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 17-7. The Contents of the Claims.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class ClaimsModel : PageModel {

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

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

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

        public IList<Claim> Claims { get; set; } = new List<Claim>();

        public string GetName(string claimType) =>
            (Uri.IsWellFormedUriString(claimType, UriKind.Absolute)
                ? System.IO.Path.GetFileName(new Uri(claimType).LocalPath)
                : claimType).ToUpper();

        public async Task OnGetAsync(string id) {
            if (id != null) {
                AppUserObject = await UserManager.FindByIdAsync(id) ?? new AppUser();
                Claims = (await UserManager.GetClaimsAsync(AppUserObject))
                    .OrderBy(c => c.Type).ThenBy(c => c.Value).ToList();
            }
        }
```

```
        public async Task<IActionResult> OnPostAdd(string id, string type,
                string value) {
            AppUser user = await UserManager.FindByIdAsync(id);
            await UserManager.AddClaimAsync(user, new Claim(type, value));
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostEdit(string id, string oldType,
                string type, string oldValue, string value) {
            AppUser user = await UserManager.FindByIdAsync(id);
            if (user != null) {
                await UserManager.ReplaceClaimAsync(user,
                    new Claim(oldType, oldValue), new Claim(type, value));
            }
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostDelete(string id, string type,
                string value) {
            AppUser user = await UserManager.FindByIdAsync(id);
            await UserManager.RemoveClaimAsync(user, new Claim(type, value));
            return RedirectToPage();
        }
    }
}
```

The page model provides its view with claims data obtained through the `UserManager<T >` class, using the methods described in Table 17-3. Changes to a user's claims are also performed using the `UserManager<T>` class, allowing claims to be created, edited, and deleted.

To integrate the claims feature into the rest of the application, add the expressions shown in Listing 17-8 to the `_UserTableRow.cshtml` file in the `Pages/Store` folder.

Listing 17-8. Adding a Button 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>
}
```

An `if` expression determines whether the user store supports claims, and a button that navigates to the `Claims` Razor Page is displayed if it does. Restart ASP.NET Core and request `http://localhost:5000/users`. Click the Claims button for Alice, and you will see the set of claims stored for that user, as shown in Figure 17-2. You can use the form to add claims or change claim types and values. Claims can be removed by clicking the Delete button.

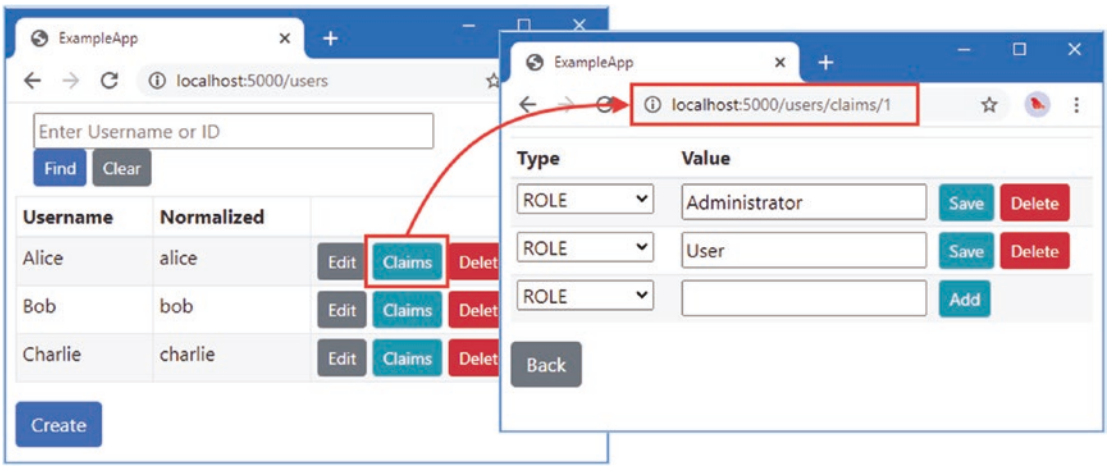

Figure 17-2. *Managing claims through the UserManager<T> class*

Storing Roles in the User Store

Even though roles can be expressed as claims, support for roles predates the adoption of claims, and roles remain the most common way to manage authorization. User stores that can manage roles implement the `IUserRoleStore<T>` interface, where `T` is the user class. The `IUserRoleStore<T>` interface defines the methods described in Table 17-5. As with user store interfaces, the methods in Table 17-5 define a `CancellationToken` parameter that is used to receive notifications when an asynchronous operation is canceled.

■ **Caution** The way that Identity manages roles presents a trap for the unwary, as explained in the "Understanding the Role Normalization Pitfall" section later in this chapter.

Table 17-5. *The IUserRoleStore<T> Methods*

Name	Description
`GetUsersInRoleAsync(role, token)`	This method returns a list of user objects representing the users who have been assigned to the specified role.
`GetRolesAsync(user, token)`	This method returns a list of the role names for the specified user.
`IsInRoleAsync(user, roleName, token)`	This method returns `true` if the specified user is a member of the specified role.
`AddToRoleAsync(user, roleName, token)`	This method adds the specified user to a role.
`RemoveFromRoleAsync(user, roleName, token)`	This method removes the specified user from a role.

To extend the user store to manage role data, add a class file named UserStoreRoles.cs to the Identity/Store folder and use it to define the partial class shown in Listing 17-9.

Listing 17-9. The Contents of the UserStoreRoles.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {
    public partial class UserStore : IUserRoleStore<AppUser> {

        public Task<IList<AppUser>> GetUsersInRoleAsync(string roleName,
                CancellationToken token)
            => GetUsersForClaimAsync(new Claim(ClaimTypes.Role, roleName), token);

        public async Task<IList<string>> GetRolesAsync(AppUser user,
                CancellationToken token)
            => (await GetClaimsAsync(user, token))
                .Where(claim => claim.Type == ClaimTypes.Role)
                .Distinct().Select(claim => Normalizer.NormalizeName(claim.Value))
                .ToList();

        public async Task<bool> IsInRoleAsync(AppUser user, string
                normalizedRoleName, CancellationToken token)
            => (await GetRolesAsync(user, token)).Any(role =>
                    Normalizer.NormalizeName(role) == normalizedRoleName);

        public Task AddToRoleAsync(AppUser user, string roleName,
                CancellationToken token)
            => AddClaimsAsync(user, GetClaim(roleName), token);

        public async Task RemoveFromRoleAsync(AppUser user,
                string normalizedRoleName, CancellationToken token) {
            IEnumerable<Claim> claimsToDelete = (await GetClaimsAsync(user, token))
                .Where(claim => claim.Type == ClaimTypes.Role
                    && Normalizer.NormalizeName(claim.Value) == normalizedRoleName);
           await RemoveClaimsAsync(user, claimsToDelete, token);
        }

        private IEnumerable<Claim> GetClaim(string role) =>
            new[] { new Claim(ClaimTypes.Role, role) };
    }
}
```

ASP.NET Core Identity doesn't dictate how a user store manages roles. Since I already have support for managing claims and roles can easily be expressed as claims, I have implemented the IUserRoleStore<T> interface by storing claims with the Role type. The only complication is that the methods that the UserManager<T> class provides for managing roles, which are described in Table 17-6 in the next section,

normalize the names of roles for consistency. This is a sensible policy but means additional work is required in Listing 17-9 to ensure that role names are mapped onto claims properly. (No seed data is required for this example because the store is already seeded with role claims.)

Managing Roles in the User Store

The `UserManager<T>` class provides a set of methods that provide access to roles when the user store class implements the `IUserRoleStore<T>` interface, as described in Table 17-6, along with a property that allows support for roles to be checked.

Table 17-6. *The UserManager<T> Members for Roles*

Name	Description
`SupportsUserRole`	This property returns true when the user store implements the `IUserRoleStore<T>` interface.
`AddToRoleAsync(user, role)`	This method adds the user to a role. The role name is normalized, and the store's `IsInRoleAsync` is used to determine if the user is already in the role, in which case an exception is thrown. Otherwise, the store's `AddToRoleAsync` method is used to add the user to the role, after which the user manager's update sequence is performed.
`AddToRolesAsync(user, roles)`	This convenience method adds the user to multiple roles using the same approach as the `AddToRoleAsync` method. The update sequence is performed once the user has been added to all of the roles.
`RemoveFromRoleAsync(user, role)`	This method removes a user from a role. The role is normalized, and the store's `IsInRoleAsync` method is used to make sure the user is a member of the role before the store's `RemoveFromRoleAsync` is called, followed by the user manager's update sequence.
`RemoveFromRolesAsync(user, roles)`	This convenience method removes the user from multiple roles using the same approach as the `RemoveFromRoleAsync` method. The user manager's update sequence is performed once the user has been removed from all of the roles.
`GetRolesAsync(user)`	This method returns the user's roles, expressed as a list that is obtained from the user store's `GetRolesAsync` method.
`IsInRoleAsync(user, role)`	This method returns `true` if the user has the specified role. The role is normalized before being passed to the user store's `IsInRoleAsync` method.
`GetUsersInRoleAsync(role)`	This method returns a list of users who have the specified role. The role is normalized before being passed to the user store's `GetUsersInRoleAsync` method.

To manage a user's roles, add a Razor Page named `UserRoles.cshtml` to the `Pages/Store` folder, with the content shown in Listing 17-10.

Listing 17-10. The Contents of the UserRoles.cshtml File in the Pages/Store Folder

```
@page "/users/roles/{id?}"
@model ExampleApp.Pages.Store.UserRolesModel

<h4 class="bg-primary text-white text-center p-2">Roles</h4>
<div class="m-2">
    <table class="table table-sm table-striped">
        <thead><tr><th>Role</th><th/></tr></thead>
        <tbody>
            @foreach (string role in Model.Roles) {
                <tr>
                    <td>@role</td>
                    <td>
                        <form method="post">
                            <input type="hidden" name="id" value="@Model.Id" />
                            <input type="hidden" name="role" value="@role" />
                            <button type="submit" class="btn btn-sm btn-danger"
                                    asp-page-handler="delete">
                                Delete
                            </button>
                        </form>
                    </td>
                </tr>
            }
            <tr>
                <td>
                    <form method="post" id="newRole">
                        <input type="hidden" name="id" value="@Model.Id" />
                        <input class="w-100" name="newRole" placeholder="Add Role" />
                    </form>
                </td>
                <td>
                    <button type="submit" class="btn btn-sm btn-primary"
                            asp-page-handler="add" form="newRole">
                        Add
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
    <div>
        <a asp-page="users" class="btn btn-secondary">Back</a>
    </div>
</div>
```

The Razor Page presents a table containing the roles to which the user has been assigned, along with a Delete button that will remove the user from the role. There is a row in the table that allows the user to be added to new roles.

Add the code shown in Listing 17-11 to the `UserRoles.cshtml.cs` file to define the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 17-11. The Contents of the UserRoles.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.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ExampleApp.Pages.Store {

    public class UserRolesModel : PageModel {

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

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

        public IEnumerable<string> Roles { get; set; } = Enumerable.Empty<string>();

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

        public async void OnGet() {
            AppUser user = await GetUser();
            if (user != null) {
                Roles = await UserManager.GetRolesAsync(user);
            }
        }

        public async Task<IActionResult> OnPostAdd(string newRole) {
            await UserManager.AddToRoleAsync(await GetUser(), newRole);
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostDelete(string role) {
            await UserManager.RemoveFromRoleAsync(await GetUser(), role);
            return RedirectToPage();
        }

        private Task<AppUser> GetUser() => Id == null
            ? null : UserManager.FindByIdAsync(Id);
    }
}
```

The page model class uses the method defined by the `UserManager<T>` class to find `AppUser` objects and manages role assignments using the methods described in Table 17-6. The GET handler method gets the current role assignments for the user; the POST handler methods add and delete role memberships.

In Listing 17-12, I have added a button to the `_UserTableRow.cshtml` partial view that navigates to the `UserRoles` Razor Page if the user store supports role data.

Listing 17-12. Adding a Button 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>
}
```

Restart ASP.NET Core, request `http://localhost:5000/users`, and click the Roles button for one of the users. The roles displayed will correspond to the user's role claims, although the names will be normalized, as shown in Figure 17-3. Bear in mind that the roles stored in the database won't be used for authorization until Chapter 18.

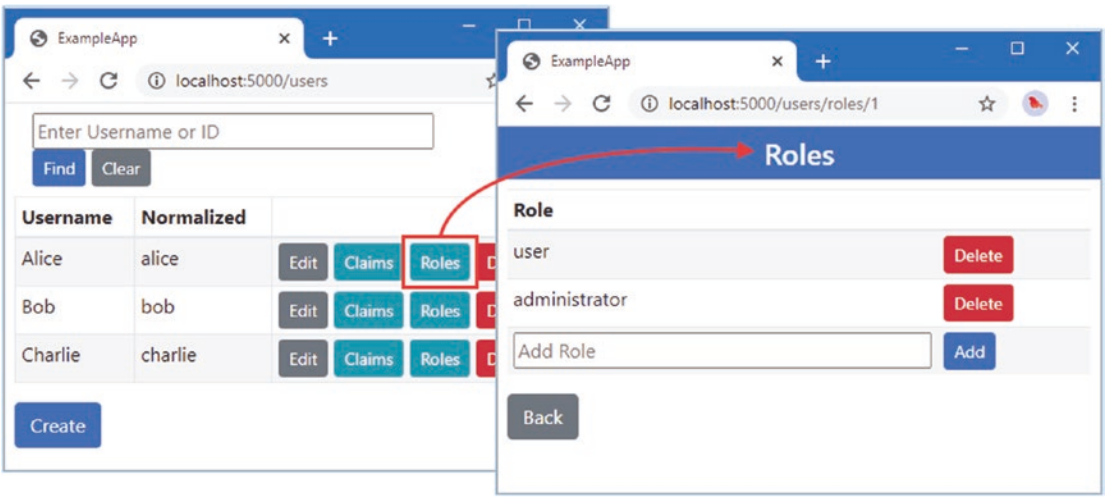

Figure 17-3. *Managing roles*

Understanding the Role Normalization Pitfall

There is a mismatch between the way that ASP.NET Core assesses roles and the way roles are stored by Identity. When a user is assigned to a role using the methods described in Table 17-6, the name of the role is normalized before it is added to the store.

When ASP.NET Core assesses role requirements specified through the `Authorize` attribute, it uses the `ClaimsPrincipal.IsInRole`, which performs a case-sensitive search for matching role claims. This can often fail to match roles whose names have been normalized in the store. You can see the effect of the normalization in Figure 17-3. Even though Alice has role claims for the `User` and `Administrator` roles, they are displayed as `user` and `administrator`.

There are three ways of avoiding this problem. The first is to express role requirements to match the normalized roles names in the store. This is a simple fix, but it only works when you can be confident that the normalizer won't change. This means that roles should be specified in uppercase (e.g., `ADMINISTRATOR`) when the default Identity normalizer is used and lowercase (e.g., `administrator`) for the normalizer I created in Chapter 16.

The second approach is to use the normalized role name assigned to a user object as a key to locate the non-normalized name in another data store. This is the approach that the Entity Framework Core user store uses and is the reason that a role store was required to use roles in Part 1, even though I didn't need any of the direct features that the role store provided (and which are described in Chapter 19).

The remaining approach is to manage roles as claims directly, avoiding the normalization performed by the `UserManager<T>` methods. None of these approaches is ideal, but I prefer to store roles as claims because it means that a future change to the normalizer doesn't break role-based authorization. In Chapter 19, I create a custom role store and demonstrate how it can be used to create a master list of roles.

In Listing 17-13, I have modified the `UserRoles` page model class to obtain roles as claims.

Listing 17-13. Using Claims in the UserRoles.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.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Security.Claims;

namespace ExampleApp.Pages.Store {

    public class UserRolesModel : PageModel {

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

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

        public IEnumerable<string> Roles { get; set; } = Enumerable.Empty<string>();

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

        public async void OnGet() {
            AppUser user = await GetUser();
            if (user != null) {
                //Roles = await UserManager.GetRolesAsync(user);
                Roles = (await UserManager.GetClaimsAsync(user))?
                    .Where(c => c.Type == ClaimTypes.Role).Select(c => c.Value);
            }
        }
```

```
        public async Task<IActionResult> OnPostAdd(string newRole) {
            //await UserManager.AddToRoleAsync(await GetUser(), newRole);
            await UserManager.AddClaimAsync(await GetUser(),
                new Claim(ClaimTypes.Role, newRole));
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostDelete(string role) {
            await UserManager.RemoveFromRoleAsync(await GetUser(), role);
            return RedirectToPage();
        }

        private Task<AppUser> GetUser() => Id == null
            ? null : UserManager.FindByIdAsync(Id);
    }
}
```

Getting the user's roles and adding a new role are performed using the methods that manage claims. I don't need to change the code that removes a role because that operation can be done using the normalized role name without causing any problems. Restart ASP.NET Core and view the roles for Alice; you will see that the role names are now mixed case, as shown in Figure 17-4.

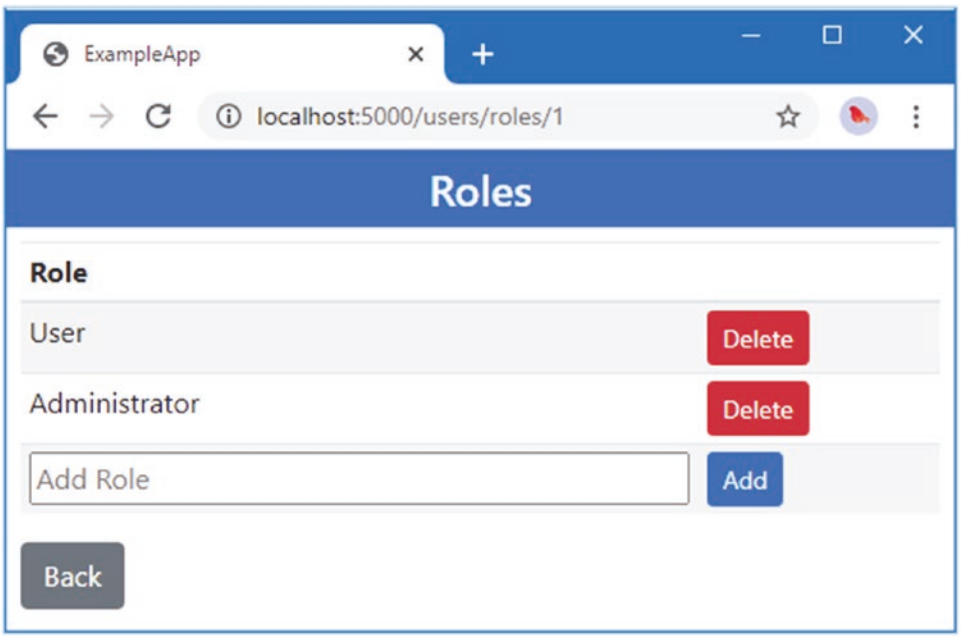

Figure 17-4. *Avoiding the role normalization pitfall*

Confirming User Contact Data

Confirmation is the process of verifying that the contact data in the store for a user is correct. A user can enter a valid phone number or email address, but it is important to confirm they can receive calls or emails before using those values, especially when they are used to sign into the application or as part of a password recovery process.

Understanding the Confirmation Process

The confirmation process is performed by generating a token and sending it to the user through the communications channel being confirmed. This means, for example, that the token is emailed to confirm an email address and sent as an SMS text message to a phone number.

The user then provides the token to the application to demonstrate they control the email address or phone number. In the case of emails, this typically involves clicking a link that includes the token, as shown in Figure 17-5. In the case of SMS messages, the user typically enters the token into an HTML form.

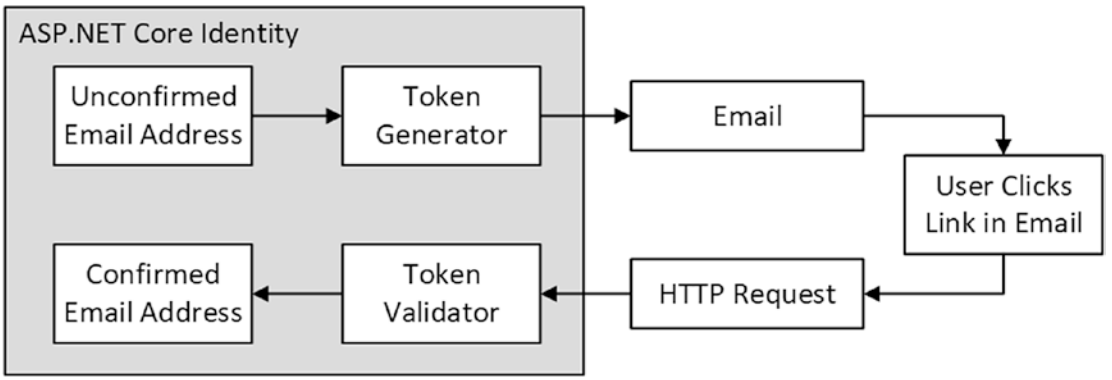

Figure 17-5. *The confirmation sequence*

Creating the Email and SMS Service Providers

The confirmation process requires the ability to send confirmation emails or texts to users, which requires the use of a third-party service or, in a corporate setting, central servers.

For this book, I am going to only simulate the process of sending emails and SMS messages. The process of configuring communications providers is a topic in its own right and beyond the scope of this book. Create the `ExampleApp/Services` folder and add to it a class file named `EmailService.cs` with the code shown in Listing 17-14.

Listing 17-14. The Contents of the EmailService.cs File in the Services Folder

```
using ExampleApp.Identity;
using System;

namespace ExampleApp.Services {
```

```
    public interface IEmailSender {

        public void SendMessage(AppUser user, string subject, params string[] body);
    }

    public class ConsoleEmailSender : IEmailSender {

        public void SendMessage(AppUser user, string subject, params string[] body) {
            Console.WriteLine("--- Email Starts ---");
            Console.WriteLine($"To: {user.EmailAddress}");
            Console.WriteLine($"Subject: {subject}");
            foreach (string str in body) {
                Console.WriteLine(str);
            }
            Console.WriteLine("--- Email Ends ---");
        }
    }
}
```

The `IEmailSender` interface defines an abstract interface for sending emails, and `ConsoleEmailSender` implements the interface and simulates sending an email by writing messages to the console.

USING EMAIL AND SMS PROVIDERS

My advice is to use commercial email and SMS services rather than try to set up your own. Getting messaging working properly can be difficult, and it is easy to misconfigure the services so that your users receive too many or too few messages. If you don't know where to start, then try Twilio (https://www.twilio.com) or Amazon's Simple Notification Service (https://aws.amazon.com/sns). Twilio is the provider that Microsoft partners with for Azure, and Amazon Web Services needs no introduction. I have no relationship with either service, but both provide a C# API and offer a range of pricing options, including free tiers. If you dislike SendGrid or Amazon Web Services, there are many alternatives from which to choose.

Next, add a class file named `SMSService.cs` to the `ExampleApp/Services` folder with the code shown in Listing 17-15.

Listing 17-15. The Contents of the SMSService.cs File in the Services Folder

```
using ExampleApp.Identity;
using System;

namespace ExampleApp.Services {

    public interface ISMSSender {

        public void SendMessage(AppUser user, params string[] body);
    }
```

```
    public class ConsoleSMSSender : ISMSSender {

        public void SendMessage(AppUser user, params string[] body) {
            Console.WriteLine("--- SMS Starts ---");
            Console.WriteLine($"To: {user.PhoneNumber}");
            foreach (string str in body) {
                Console.WriteLine(str);
            }
            Console.WriteLine("--- SMS Ends ---");
        }
    }
}
```

The ISMSSender interface defines an abstract interface for sending SMS messages, and ConsoleSMSSender implements the interface and simulates sending an SMS by writing messages to the console. Listing 17-16 registers the email and SMS classes as services so they can be consumed through dependency injection.

Listing 17-16. Registering Services in the Startup.cs File in the ExampleApp Folder

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

namespace ExampleApp {
    public class Startup {

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

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

            services.AddAuthentication(opts => {
                opts.DefaultScheme
                    = CookieAuthenticationDefaults.AuthenticationScheme;
            }).AddCookie(opts => {
                opts.LoginPath = "/signin";
                opts.AccessDeniedPath = "/signin/403";
            });
```

```
            services.AddAuthorization(opts => {
                AuthorizationPolicies.AddPolicies(opts);
            });
            services.AddRazorPages();
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseRouting();
            //app.UseMiddleware<RoleMemberships>();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapRazorPages();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapFallbackToPage("/Secret");
            });
        }
    }
}
```

Storing Security Stamps in the User Store

The next step is to implement the optional interface for the user store that adds support for managing security stamps. A security stamp is a token that is updated every time a change is made to a user, which prevents confirmation tokens from being validated if there have been subsequent changes to the user's details. This avoids the situation where a user receives a confirmation token, changes their email address again, and only then validates the token, with the result that the user's ability to receive emails at the new address isn't properly validated. Security stamps are strings, and Listing 17-17 updates the user class to add a property for a stamp.

Listing 17-17. 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; }
    }
}
```

User stores that can manage security stamps implement the `IUserSecurityStampStore<T>` interface, which defines the methods described in Table 17-7. As with earlier examples, the type `T` is the user class, and the `token` parameter is a `CancellationToken` object that is used when asynchronous tasks are cancelled.

Table 17-7. *The IUserSecurityStampStore<T> Methods*

Name	Description
`SetSecurityStampAsync(user, stamp, token)`	This method sets a new security stamp for the specified user.
`GetSecurityStampAsync(user, token)`	This method retrieves the specified user's security stamp.

To extend the example user store to implement the interface, add a class file named `UserStoreSecurityStamps.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 17-18.

Listing 17-18. The Contents of the UserStoreSecurityStamps.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 : IUserSecurityStampStore<AppUser> {

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

        public Task SetSecurityStampAsync(AppUser user, string stamp,
                CancellationToken token) {
            user.SecurityStamp = stamp;
            return Task.CompletedTask;
        }
    }
}
```

The interface implementation is simple and has only to read and write the SecurityStamp property defined in Listing 17-17.

Updating the Security Stamp

The security stamp for a user object can be accessed directly through the property added in Listing 17-17 or through the UserManager<T> members described in Table 17-8.

Table 17-8. *The UserManager<T> Members for Security Stamps*

Name	Description
SupportsUserSecurityStamp	This property returns true if the user store implements the IUserSecurityStampStore<T> interface.
GetSecurityStampAsync(user)	This method returns the current security stamp for the specified user by calling the user store's GetSecurityStampAsync method.
UpdateSecurityStampAsync(user)	This method updates the security stamp for the specified user. The stamp is generated by the UserManager<T> class using random data passed to the user store's SetSecurityStampAsync method, after which the user manager's update sequence is performed.

Add a Razor View named _EditUserSecurityStamp.cshtml to the Pages/Store folder with the content shown in Listing 17-19.

Listing 17-19. The Contents of the _EditUserSecurityStamp.cshtml File in the Pages/Store Folder

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

@if (UserManager.SupportsUserSecurityStamp) {
    <tr>
        <td>Security Stamp</td>
        <td>@Model.SecurityStamp</td>
    </tr>
}
```

Add the element shown in Listing 17-20 to incorporate the new partial view into the application.

Listing 17-20. Displaying the Security Stamp 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="_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>
```

The current stamp is displayed when the user store supports security stamps. The `UserManager<T>` methods that set property values automatically update the security stamp, but I have to perform this task directly since I am setting user object properties directly, as shown in Listing 17-21.

Listing 17-21. Updating the Security Stamp in the EditUser.cshtml.cs File in the Pages/Identity Folder

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

The `CreateAsync` method automatically generates a security stamp, which is why I only update the stamp explicitly when I use the `UpdateAsync` method and when one or more properties are changed. In Listing 17-22, I have updated the code that seeds the user store to include security stamps in the initial data.

Listing 17-22. Seeding with Security Stamps in the UserStore.cs File in the Identity/Store Folder

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

Users that are created through the `UserManager<T>` class will have security stamps generated automatically, but this additional step is required for the seed users to prevent exceptions when the security stamp is changed.

To make sure that security stamps are working, restart ASP.NET Core, request `http://localhost:5000/users/edit/1`, and enter a new value into the Hobby field. Click the Save button and then click the Edit button for the Alice user, and you will see the new security stamp, as shown in Figure 17-6. A new security stamp is generated each time you make a change.

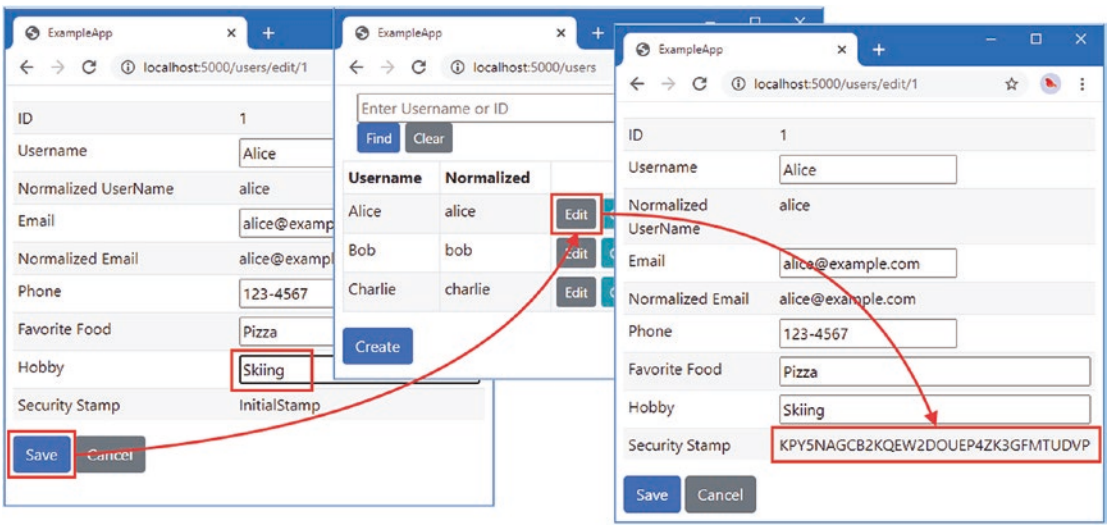

Figure 17-6. *Updating a security stamp*

Creating a Confirmation Token Generator

The security token that is sent to the user is produced by a *token generator*. Identity provides a set of default token generators, but creating a custom generator helps demonstrate how the process works. Token generators implement the `IUserTwoFactorTokenProvider<T>` interface, where `T` is the user class. The interface defines the methods described in Table 17-9.

Table 17-9. *The IUserTwoFactorTokenProvider<T> Methods*

Name	Description
`CanGenerateTwoFactorTokenAsync(manager, token)`	This method returns `true` if the generator can produce a token for the specified `UserManager<T>` and user object.
`GenerateAsync(purpose, manager, user)`	This method generates a string confirmation token for the specified purpose for a given user and user manager.
`ValidateAsync(purpose, token, manager, user)`	This method validates a token that has been generated for the specified person, user, and user manager. The method returns `true` if the token is valid and `false` otherwise.

There are no restrictions on how tokens should be generated and validated, but tokens should be simple, easy to process, and not easily altered (so that a token for one user cannot be edited to become a token to confirm details from another user, for example).

The default Identity token generator creates *time-based one-time passwords* (TOTPs), which are often used as part of a two-factor sign-in process, as described in Chapters 20 and 21.

For this chapter, I am going to take a simpler approach that will let me explain the process of confirming user details without the complexity of generating secure tokens. As with many of the examples in this part of the book, you should not use this approach in real projects and should instead rely on the tokens generators that Identity provides. To create the custom token generator, add a class file named `SimpleTokenGenerator.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 17-23.

Listing 17-23. The Contents of the SimpleTokenGenerator.cs File in the Identity Folder

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

namespace ExampleApp.Identity {
    public abstract class SimpleTokenGenerator :
            IUserTwoFactorTokenProvider<AppUser> {

        protected virtual int CodeLength { get; } = 6;

        public virtual Task<bool> CanGenerateTwoFactorTokenAsync(
            UserManager<AppUser> manager, AppUser user) =>
                Task.FromResult(manager.SupportsUserSecurityStamp);

        public virtual Task<string> GenerateAsync(string purpose,
                UserManager<AppUser> manager, AppUser user)
            => Task.FromResult(GenerateCode(purpose, user));
```

```
        public virtual Task<bool> ValidateAsync(string purpose, string token,
                UserManager<AppUser> manager, AppUser user)
            => Task.FromResult(GenerateCode(purpose, user).Equals(token));

        protected virtual string GenerateCode(string purpose, AppUser user) {
            HMACSHA1 hashAlgorithm =
                new HMACSHA1(Encoding.UTF8.GetBytes(user.SecurityStamp));
            byte[] hashCode = hashAlgorithm.ComputeHash(
                Encoding.UTF8.GetBytes(GetData(purpose, user)));
            return BitConverter.ToString(hashCode[^CodeLength..]).Replace("-", "");
        }

        protected virtual string GetData(string purpose, AppUser user)
            => $"{purpose}{user.SecurityStamp}";
    }

    public class EmailConfirmationTokenGenerator : SimpleTokenGenerator {

        protected override int CodeLength => 12;

        public async override Task<bool> CanGenerateTwoFactorTokenAsync(
                UserManager<AppUser> manager, AppUser user) {
            return await base.CanGenerateTwoFactorTokenAsync(manager, user)
                && !string.IsNullOrEmpty(user.EmailAddress)
                && !user.EmailAddressConfirmed;
        }
    }

    public class PhoneConfirmationTokenGenerator : SimpleTokenGenerator {

        protected override int CodeLength => 3;

        public async override Task<bool> CanGenerateTwoFactorTokenAsync(
                UserManager<AppUser> manager, AppUser user) {
            return await base.CanGenerateTwoFactorTokenAsync(manager, user)
                && !string.IsNullOrEmpty(user.PhoneNumber)
                && !user.PhoneNumberConfirmed;
        }
    }
}
```

I have defined a base class that is used as the base for classes that generate tokens suitable for inclusion conforming email addresses and phone numbers. The tokens are generated as hash codes of strings that include the purpose of the token, the address of number being confirmed, and the user's current security stamp.

```
...
protected virtual string GetData(string purpose, AppUser user)
    => $"{purpose}{user.SecurityStamp}";
...
```

The inclusion of the purpose ensures that a token is valid for only one type of confirmation. The inclusion of the security stamp ensures that tokens are invalidated when a change is made to the user object, ensuring that the token can be used only until a change is made in the user store.

The email and SMS confirmation tokens are different lengths. This is not a requirement, but users who receive tokens over SMS will often have to type them into an HTML form, and it is important to make that process easy. Listing 17-24 registers the token generators with ASP.NET Core Identity.

Listing 17-24. Registering a Token Generator 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.AddIdentityCore<AppUser>(opts => {
        opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
        opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
    })
    .AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
    .AddTokenProvider<PhoneConfirmationTokenGenerator>(
         TokenOptions.DefaultPhoneProvider);

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

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

The `AddTokenProvider` extension method is used to register a token generator, along with a name by which it will be known. The options pattern is used to select a token generator for a particular purpose. The `IdentityOptions.Tokens` property returns a `TokenOptions` object, which defines the properties shown in Table 17-10 for specifying token generators.

Table 17-10. *The TokenOptions Properties for Confirmation Token Generators*

Name	Description
`DefaultProvider`	This property specifies the name of the token generator that will be used by default. It is set to `Default`.
`DefaultEmailProvider`	This property specifies the name of the token generator used for email confirmations. It is set to `Email`.
`DefaultPhoneProvider`	This property specifies the name of the token generator used for phone confirmations. It is set to `Phone`.
`ChangeEmailTokenProvider`	This property specifies the token generator used for email changes. It uses `DefaultProvider` by default. (See the explanation after the table for more information about this property.)
`EmailConfirmationTokenProvider`	This property specifies the token generator used for email changes. It uses `DefaultProvider` by default. (See the explanation after the table for more information about this property.)
`ChangePhoneNumberTokenProvider`	This property specifies the token generator used for phone number changes. It uses `DefaultPhoneProvider` by default.
`PasswordResetTokenProvider`	This property specifies the token generator for confirming password changes.

When registering a token generator, you use the `AddTokenProvider` method to register the class with a name. The idea is that you can use `TokenOptions.DefaultPhoneProvider`, for example, and your generator will be used as the default generator for phone number confirmations, like this:

```
...
.AddTokenProvider<PhoneConfirmationTokenGenerator>(
    TokenOptions.DefaultPhoneProvider);
...
```

Unfortunately, the `TokenOptions` class uses the `DefaultProvider` property as the value for the `ChangeEmailTokenProvider` and `EmailConfirmationTokenProvider` configuration options. This means you must use a custom name for your token generator and perform the additional step of setting the `ChangeEmailTokenProvider` and `EmailConfirmationTokenProvider` options, like this:

```
...
services.AddIdentityCore<AppUser>(opts => {
    opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
    opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
})
...
```

There are two configuration properties for email token generators, and it is a good idea to set both. I explain when each generator property is used in the next section.

Creating the Confirmation Workflow

The UserManager<T> class provides methods for generating and validating confirmation tokens, as described in Table 17-11.

Table 17-11. *The UserManager<T> Methods for Generating and Validating Tokens*

Name	Description
GenerateUserTokenAsync(user, provider, purpose)	This method uses the named provider to generate a token for the specified user and purpose.
VerifyUserTokenAsync(user, provider, purpose, token)	This method returns true if the named token generator validates the token for the specified user and purpose.

These methods are useful when you are working directly with the properties defined by the user class. The UserManager<T> class also provides convenience methods that simplify the confirmation process, as described in Table 17-12.

Table 17-12. *The UserManager<T> Email Confirmation Convenience Methods*

Name	Description
GenerateEmailConfirmationTokenAsync(user)	This method generates a token for the specified user with the EmailConfirmationTokenProvider token generator, using the email address currently in the user store.
ConfirmEmailAsync(user, token)	This method validates a token for the specified user using the EmailConfirmationTokenProvider generator. If the token is valid, the email confirmation property is set to true. The user's email property is not changed.
GenerateChangeEmailTokenAsync(user, email)	This method generates a token for the specified user with the ChangeEmailTokenProvider token generator. The user store is not modified.
ChangeEmailAsync(user, email, token)	This method validates a token for the specified user using the ChangeEmailTokenProvider generator. If the token is valid, the email address is updated, the email confirmation property is set to true, the security stamp is updated, and the user manager's update sequence is performed.

The GenerateEmailConfirmationTokenAsync and ConfirmEmailAsync methods are used to confirm the email address that is already in the store, which can be useful if you have a self-service registration process and you want to confirm the user's details when a new account is created. The GenerateChangeEmailTokenAsync and ChangeEmailAsync methods are used to confirm new email addresses so that the user store isn't updated until the email address has been confirmed. Note that each pair of methods uses a distinct combination of token generator and token purpose, which means that ChangeEmailAsync method can't be used to validate tokens generated by the GenerateEmailConfirmationTokenAsync method. Table 17-13 describes the set of UserManager<T> methods for confirming phone numbers.

Table 17-13. *The UserManager<T> Phone Confirmation Convenience Methods*

Name	Description
`GenerateChangePhoneNumberTokenAsync(user, phone)`	This method generates a token for the specified user with the `ChangePhoneNumberTokenProvider` token generator, using the specified phone number.
`VerifyChangePhoneNumberTokenAsync(user, token, phone)`	This method returns true if the specified token is valid for the specified phone number.
`ChangePhoneNumberAsync(user, phone, token)`	This convenience method validates a token for the specified user. If the token is valid, the phone number is updated, and the phone confirmation property is set to `true`, after which the security stamp is updated, and the user manager's update sequence is performed.

To support email and phone confirmations in the example application, add a Razor Page named `EmailPhoneChange.cshtml` to the `Pages/Store` folder with the content shown in Listing 17-25.

Listing 17-25. The Contents of the EmailPhoneChange.cshtml File in the Pages/Store Folder

```
@page "/change/{id}/{dataType}"
@model ExampleApp.Pages.Store.EmailPhoneChangeModel

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

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>
                @Model.LabelText
            </label>
            <input class="form-control" name="dataValue"
                value="@Model.CurrentValue" />
        </div>
        <button type="submit" class="btn btn-primary">Change</button>
        <a href="@($"/users/edit/{Model.AppUser.Id}")" class="btn btn-secondary">
            Cancel
        </a>
    </form>
</div>
```

The Razor Page presents a simple HTML form that allows a new value to be entered for either the email address or the phone number, which is determine from the URL. To provide the data for the form and to generate the confirmation tokens, add the code shown in Listing 17-26 to the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 17-26. The Contents of the EmailPhoneChange.cshtml.cs File in the Pages/Store Folder

```
using ExampleApp.Identity;
using ExampleApp.Services;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
```

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

namespace ExampleApp.Pages.Store {

    public class EmailPhoneChangeModel : PageModel {

        public EmailPhoneChangeModel(UserManager<AppUser> manager,
                IEmailSender email, ISMSSender sms) {
            UserManager = manager;
            EmailSender = email;
            SMSSender = sms;
        }

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

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

        public bool IsEmail => DataType.Equals("email");

        public AppUser AppUser { get; set; }

        public string LabelText => DataType ==
            "email" ? "Email Address" : "Phone Number";

        public string CurrentValue => IsEmail
             ? AppUser.EmailAddress : AppUser.PhoneNumber;

        public async Task OnGetAsync(string id, string data) {
            AppUser = await UserManager.FindByIdAsync(id);
        }

        public async Task<IActionResult> OnPost(string id, string dataValue) {
            AppUser = await UserManager.FindByIdAsync(id);
            if (IsEmail) {
                string token = await UserManager
                    .GenerateChangeEmailTokenAsync(AppUser, dataValue);
                EmailSender.SendMessage(AppUser, "Confirm Email",
                    "Please click the link to confirm your email address:",
                     $"http://localhost:5000/validate/{id}/email/{dataValue}:{token}");
            } else {
                string token = await UserManager
                    .GenerateChangePhoneNumberTokenAsync(AppUser, dataValue);
                SMSSender.SendMessage(AppUser,
                    $"Your confirmation token is {token}");
            }
```

```
            return RedirectToPage("EmailPhoneConfirmation",
                new { id = id, dataType = DataType, dataValue = dataValue });
        }
    }
}
```

The GET page handler method locates the user whose data is being modified so that the current values can be displayed in the HTML form. The POST page handler method generates confirmation tokens and generates either an email or SMS message that provides the user with the token, after which a redirection is performed.

To provide the confirmation step, add a Razor Page named `EmailPhoneConfirmation.cshtml` to the `Pages/Store` folder with the content shown in Listing 17-27.

Listing 17-27. The Contents of the EmailPhoneConfirmation.cshtml File in the Pages/Store Folder

```
@page "/validate/{id}/{dataType}/{dataValue?}"
@model ExampleApp.Pages.Store.EmailPhoneConfirmationModel

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

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

<div class="m-2">
    <form method="post">
        <input type="hidden" name="id" value="@Model.AppUser.Id" />
        <input type="hidden" name="dataValue" value="@Model.DataValue" />
        <div class="form-group">
            <label>
                Enter Confirmation Token
            </label>
            <input class="form-control" name="token" />
        </div>
        <button type="submit" class="btn btn-primary">Confirm</button>
        <a href="@($"/users/edit/{Model.AppUser.Id}")" class="btn btn-secondary">
            Cancel
        </a>
    </form>
</div>
```

The page displays an `input` element into which the user can enter a confirmation code. To complete the confirmation process, add the code shown in Listing 17-28 to the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 17-28. The Contents of the EmailPhoneConfirmation.cshtml.cs File in the Pages/Store Folder

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

```
namespace ExampleApp.Pages.Store {

    public class EmailPhoneConfirmationModel : PageModel {

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

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

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

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

        public bool IsEmail => DataType.Equals("email");

        public AppUser AppUser { get; set; }

        public async Task<IActionResult> OnGetAsync(string id) {
            AppUser = await UserManager.FindByIdAsync(id);
            if (DataValue != null && DataValue.Contains(':')) {
                string[] values = DataValue.Split(":");
                return await Validate(values[0], values[1]);
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(string id,
                string token, string dataValue) {
            AppUser = await UserManager.FindByIdAsync(id);
            return await Validate(dataValue, token);
        }

        private async Task<IActionResult> Validate(string value, string token) {
            IdentityResult result;
            if (IsEmail) {
                result = await UserManager.ChangeEmailAsync(AppUser, value, token);
            } else {
                result = await UserManager.ChangePhoneNumberAsync(AppUser, value,
                    token);
            }
            if (result.Succeeded) {
                return Redirect($"/users/edit/{AppUser.Id}");
            } else {
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError(string.Empty, err.Description);
                }
                return Page();
            }
        }
    }
}
```

The page model class will validate tokens that are received through the HTML form and received through the URL, which will allow users to click the links in the emails they receive.

The final step is to add buttons that will allow the user's email address and phone numbers to be changed. For email, make the changes shown in Listing 17-29 to the _EditUserEmail partial view.

Listing 17-29. Confirming Email Changes in the _EditUserEmail.cshtml File in the Pages/Store Folder

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

@if (UserManager.SupportsUserEmail) {
    <tr>
        <td>Email</td>
        <td>
            @if (await UserManager.FindByIdAsync(Model.Id) == null) {
                <input class="w-OO" asp-for="EmailAddress" />
            } else {
                @Model.EmailAddress
                <a asp-page="EmailPhoneChange" asp-route-id="@Model.Id"
                   asp-route-datatype="email"
                   class="btn btn-sm btn-secondary align-top">Change</a>
            }
        </td>
    </tr>
    <tr>
        <td>Normalized Email</td>
        <td>
            @(Model.NormalizedEmailAddress?? "(Not Set)")
            <input type="hidden" asp-for="NormalizedEmailAddress" />
            <input type="hidden" asp-for="EmailAddressConfirmed" />
        </td>
    </tr>
}
```

For phone numbers, make the changes shown in Listing 17-30 to the _EditUserPhone partial view.

Listing 17-30. Confirming Phone Changes in the _EditUserPhone.cshtml File in the Pages/Store Folder

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

@if (UserManager.SupportsUserPhoneNumber) {
    <tr>
        <td>Phone</td>
        <td>
            @if (await UserManager.FindByIdAsync(Model.Id) == null) {
                <input class="w-OO" asp-for="PhoneNumber" />
            } else {
                @Model.PhoneNumber
                <a asp-page="EmailPhoneChange" asp-route-id="@Model.Id"
                   asp-route-datatype="phone"
                   class="btn btn-sm btn-secondary align-top">Change</a>
```

```
            }
            <input type="hidden" asp-for="PhoneNumberConfirmed" />
        </td>
    </tr>
}
```

Restart ASP.NET Core, and request `http://localhost:5000/users`. Click the Edit button for the Alice user and click the Change button to edit the email address. If you look at the output produced by ASP.NET Core, you will see a simulated email message like this:

```
--- Email Starts ---
To: alice@example.com
Subject: Confirm Email
Please click the link to confirm your email address:
http://localhost:5000/validate/1/email/alicesmith@example.com:CAD4E404EED9129055BF41D4
--- Email Ends ---
```

You will see the same confirmation code as shown earlier because the security stamp is set to a fixed value when the user store is seeded. Subsequent changes will produce different codes because the stamp will change randomly. You can navigate to the URL shown in the email or enter the validation code directly (the code follows the : character in the path and is `CAD4E404EED9129055BF41D4` in this example). Once the change has been confirmed, the updated email address is displayed, as shown in Figure 17-7.

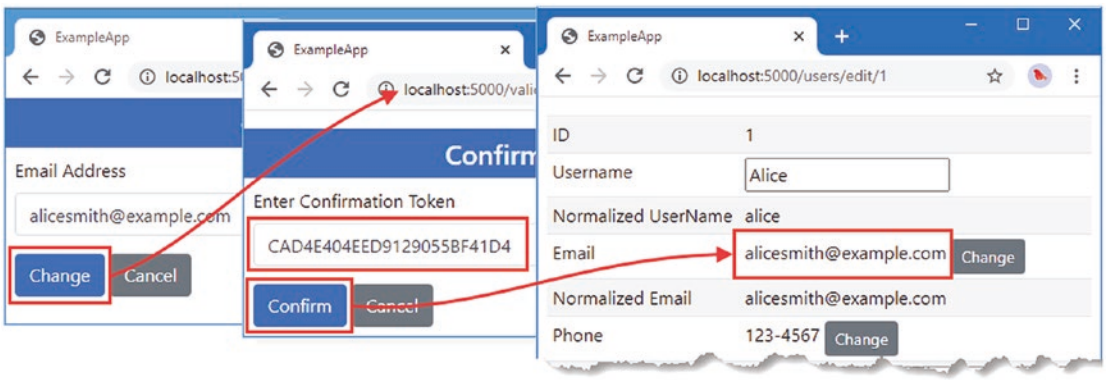

Figure 17-7. *Changing an email address*

Click the Change button next to the user's phone number, enter a new number and click the Change button. A simulated SMS message will be displayed in the console output, like this:

```
--- SMS Starts ---
To: 123-4567
Your confirmation token is B1278F
--- SMS Ends ---
```

Copy the code displayed in your message—which will be different from the one shown earlier—into the text field and click the Confirm button. The confirmation token will be validated, and the modified user data will be displayed, as shown in Figure 17-8.

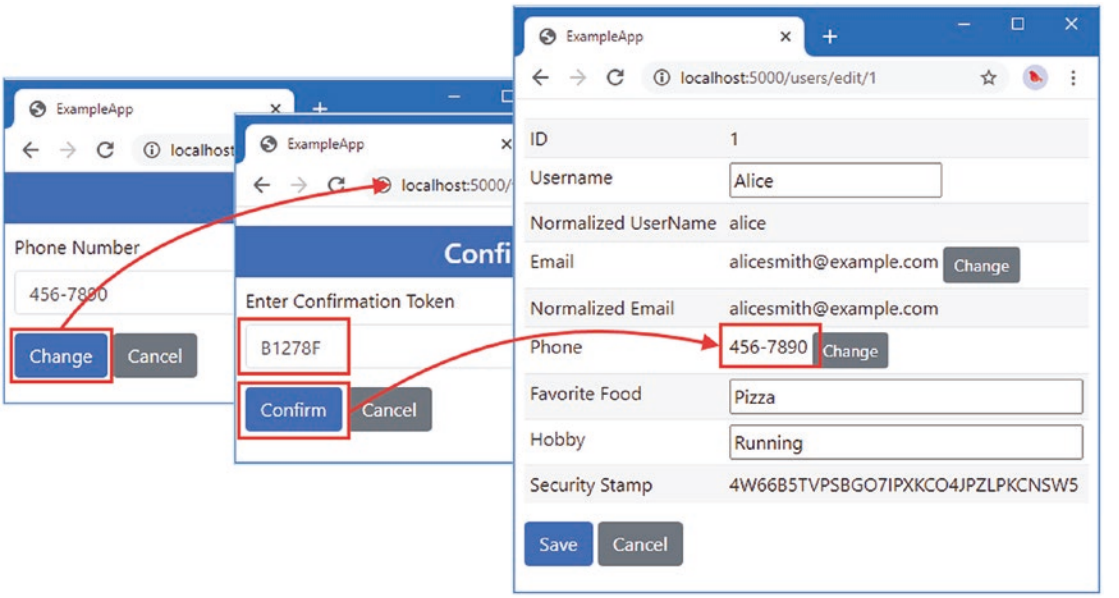

Figure 17-8. *Changing a phone number*

The validation features still applied to the data but not until after a confirmation token has been validated. You can see this by attempting to make a change to an email address outside of the `example.com` domain, which is the restriction imposed by the custom validator defined in Chapter 16. When you enter the confirmation code (or navigate to the URL given in the email), you will see a validation error, as shown in Figure 17-9, and the email address will not be updated.

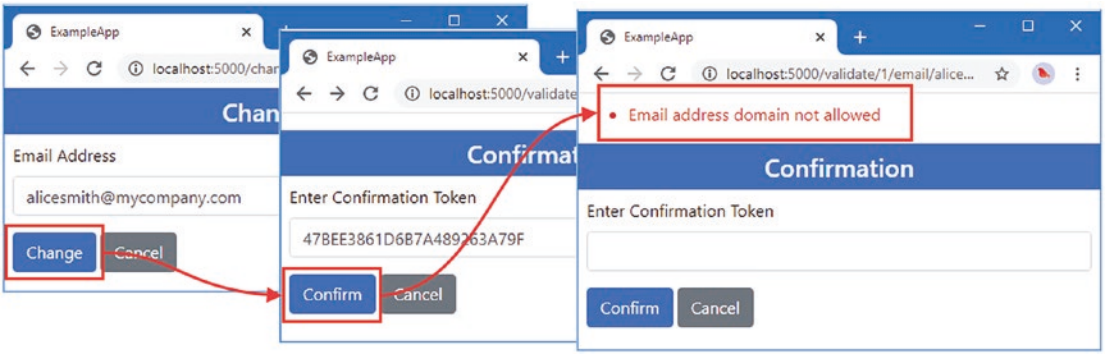

Figure 17-9. *A validation error when changing an email address*

You will also receive a validation error if you enter a different token from the one specified in the message, change the URL to specify a different email address, or alter the user object so that it contains a different security stamp. All of these actions will cause the verification process to fail because the token received from the user won't match the one generated by the application during the validation process.

Summary

In this chapter, I extended the user store to support roles and claims and set up the confirmation process for email addresses and phone numbers. In the next chapter, I extend the user store to support passwords so that users can sign into the application.