



# Creating a User Store

In this chapter, I add Identity to the example project and create a custom user store. In Part 1, I used the default user store that Microsoft provides, which uses Entity Framework Core to store user data in a relational database, and this is the store you should use in real projects.

The user store I create in this chapter stores its data in memory, which makes it easy to explain how user stores work without getting bogged down in how to serialize and persist the data. Table 16-1 puts the user store in context.

**Table 16-1.** *Putting the User Store in Context*

Question	Answer
What is it?	The user store is the data repository for Identity data.
Why is it useful?	The user store maintains the data that Identity manages. Without a user store, none of the features that Identity provides would be possible.
How is it used?	The user store is registered as a service and consumed by Identity through the ASP.NET Core dependency injection feature.
Are there any pitfalls or limitations?	The main issue with custom user stores is ensuring that each optional interface is implemented without impacting any of the others. As you will see in later examples, some features are closely related, and care must be taken to ensure consistent results.
Are there any alternatives?	A user store is required to use Identity, but you do not have to create a custom implementation.

Table 16-2 summarizes the chapter.

**Table 16-2.** Chapter Summary

Problem	Solution	Listing
Define a custom user class	Create a class that has a unique key property and can store regular and normalized versions of a name. Define additional properties to support the optional features implemented by the user store.	7-8, 24, 31
Define a custom user store	Create an implementation of the IUserStore<T> interface. Additional features can be supported by implementing optional interfaces. Register the store as a service.	11, 14, 22, 25-27
Define a custom normalizer	Create an implementation of the ILookupNormalizer interface. Register the normalizer as a service.	12-14
Access the user store	Use the members of the UserManager<T> class.	15-21, 23, 28-30
Validate user data before it is added to the store	Create an implementation of the IUserValidator<T> method and register it as a service.	35, 26

## Preparing for This Chapter

This chapter uses the ExampleApp project from Chapter 15. To prepare for this chapter, replace the Startup class with the code shown in Listing 16-1, which removes the custom authorization middleware used in the previous chapter and replaces it with the built-in authorization. I have also enabled the custom RoleMemberships middleware component as a source of user claims and removed the application model conventions for Razor Pages and the MVC Framework. The MapFallbackToPage method is used to select the Secret Razor Page for requests that are not matched by another endpoint.

**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 16-1.** The Contents of 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;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {

            services.AddTransient<IAuthorizationHandler,
                CustomRequirementHandler>();
```

```

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<AuthorizationReporter>();
    app.UseMiddleware<RoleMemberships>();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        //endpoints.MapGet("/", async context => {
        //    await context.Response.WriteAsync("Hello World!");
        //});
        endpoints.MapRazorPages();
        endpoints.MapDefaultControllerRoute();
        endpoints.MapFallbackToPage("/Secret");
    });
}
}
}

```

Disable the custom fallback authorization policy and remove the reference to the OtherScheme from the UsersExceptBob policy, as shown in Listing 16-2.

**Listing 16-2.** Altering Authorization Policies in the AuthorizationPolicies.cs File in the Custom Folder

```

using Microsoft.AspNetCore.Authorization;
using System.Linq;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ExampleApp.Custom {

    public static class AuthorizationPolicies {

        public static void AddPolicies(AuthorizationOptions opts) {
            //opts.FallbackPolicy = new AuthorizationPolicy(
            //    new IAuthorizationRequirement[] {
            //        new RolesAuthorizationRequirement(

```

```

//          new [] { "User", "Administrator" }},
//          new AssertionRequirement(context =>
//              !string.Equals(context.User.Identity.Name, "Bob"))
//      }, new string[] { "TestScheme" });

opts.AddPolicy("UsersExceptBob", builder =>
    builder.RequireRole("User")
    .AddRequirements(new AssertionRequirement(context =>
        !string.Equals(context.User.Identity.Name, "Bob"))));
//.AddAuthenticationSchemes("OtherScheme"));

opts.AddPolicy("NotAdmins", builder =>
    builder.AddRequirements(new AssertionRequirement(context =>
        !context.User.IsInRole("Administrator"))));
}
}
}

```

Remove the `Authorize` attribute from the `Protected` action of the `Home` controller, as shown in Listing 16-3.

**Listing 16-3.** Removing an Attribute in the `HomeController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace ExampleApp.Controllers {

    [Authorize]
    public class HomeController : Controller {

        public IActionResult Test() => View();

        //[Authorize(Roles = "User", AuthenticationSchemes = "OtherScheme")]
        public IActionResult Protected() => View("Test", "Protected Action");

        [AllowAnonymous]
        public IActionResult Public() => View("Test", "Unauthenticated Action");
    }
}

```

Finally, disable the partial view that displays the authorization results used in Chapter 15, as shown in Listing 16-4.

**Listing 16-4.** Removing the Partial View in the `_Layout.cshtml` File in the `Pages/Shared` Folder

```

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ExampleApp</title>

```

```

    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div>
        @RenderBody()
    </div>
    @*<partial name="_AuthorizationReport" />*@
</body>
</html>

```

The effect of these changes is to restore the built-in authorization features so that users can be signed in at the /signin URL using the Cookie authentication scheme. The custom RoleMemberships middleware component provides a source of user claims that are used by the authorization policies. Run the command shown in Listing 16-5 in the ExampleApp folder to start ASP.NET Core.

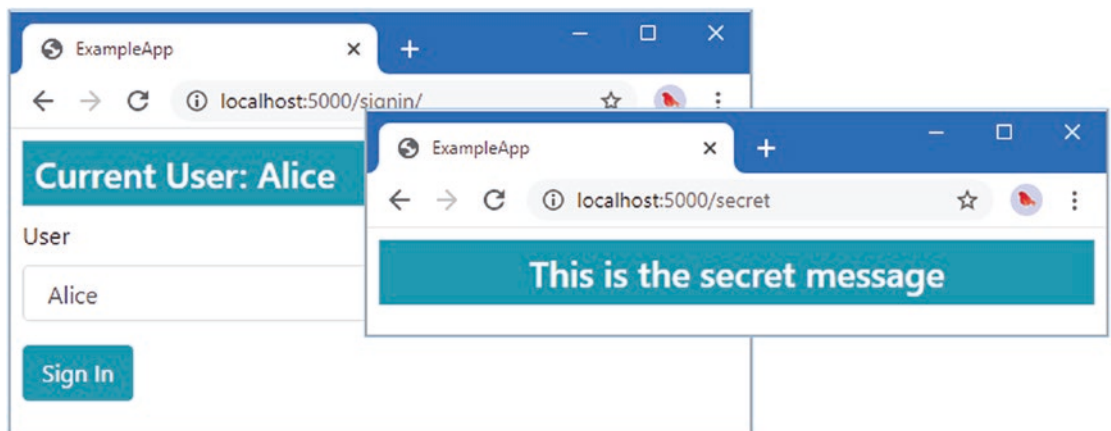
**Listing 16-5.** Starting ASP.NET Core

---

```
dotnet run
```

---

Request `http://localhost:5000/signin`, select Alice from the list, and click the Sign In button. Request `http://localhost:5000/secret`, and the request will be authenticated using a cookie and granted access to the Secret Razor Page, as shown in Figure 16-1.



**Figure 16-1.** Running the example application

## Installing ASP.NET Core Identity

Use a command prompt to run the command shown in Listing 16-6 in the ExampleApp folder to add the core Identity package to the project.

**Listing 16-6.** Adding the Core Identity Package to the Project

---

```
dotnet add package Microsoft.Extensions.Identity.Core --version 5.0.0
```

---

## Creating an Identity User Store

In earlier chapters, I hard-coded the list of users into the application, which can get you started but quickly becomes difficult to manage and requires a new release every time a user is added or removed. A user store provides a consistent way to manage user data.

### Creating the User Class

The first step in creating a user store is to define the user class, instances of which will be used to represent users in the application. No specific base class is required, but the instances of the class must be distinguishable from one another and must be able to store basic information about the user. To define the user class for this chapter, create the `ExampleApp/Identity` folder and add to it a class file named `AppUser.cs` with the code shown in Listing 16-7.

**Listing 16-7.** The Contents of the `AppUser.cs` File in the Identity Folder

```
using System;

namespace ExampleApp.Identity {
    public class AppUser {

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

        public string UserName { get; set; }

        public string NormalizedUserName { get; set; }
    }
}
```

The `Id` property will be assigned a unique identifier representing a user and defaults to a GUID value. The `UserName` property will store the user's account name in the application. The `NormalizedUserName` contains a normalized representation of the `UserName` value, which I explain in the next section. This is a minimal user class to which I will add properties as the user store develops.

I need to be able to easily copy values from one `AppUser` object to another. This will make it easier to implement the store in the way that Identity expects so that changes to an instance of the user class will be discarded if they are not explicitly saved to the user store. Add a class file named `StoreClassExtentions.cs` to the Identity folder and use it to define the extension methods shown in Listing 16-8.

**Listing 16-8.** The Contents of the `StoreClassExtentions.cs` File in the Identity Folder

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```

namespace ExampleApp.Identity {

    public static class StoreClassExtentions {

        public static T UpdateFrom<T>(this T target, T source) {
            UpdateFrom(target, source, out bool discardValue);
            return target;
        }

        public static T UpdateFrom<T>(this T target, T source, out bool changes) {
            object value;
            int changeCount = 0;
            Type classType = typeof(T);
            foreach (var prop in classType.GetProperties()) {
                if (prop.PropertyType.IsGenericType &&
                    prop.PropertyType.GetGenericTypeDefinition()
                        .Equals(typeof(ICollection))) {
                    Type listType = typeof(List<>).MakeGenericType(prop.PropertyType
                        .GetGenericArguments()[0]);
                    IList sourceList = prop.GetValue(source) as IList;
                    if (sourceList != null) {
                        prop.SetValue(target, Activator.CreateInstance(listType,
                            sourceList));
                    }
                } else {
                    if ((value = prop.GetValue(source)) != null
                        && !value.Equals(prop.GetValue(target))) {
                        classType.GetProperty(prop.Name).SetValue(target, value);
                        changeCount++;
                    }
                }
            }
            changes = changeCount > 0;
            return target;
        }

        public static T Clone<T>(this T original) =>
            Activator.CreateInstance<T>().UpdateFrom(original);
    }
}

```

I have defined methods with generic type parameters so that I can use the same code to handle different classes used by Identity. The `UpdateFrom` method will copy the value of any non-null property from one object to another, while the `Clone` method will create a copy of an object. The code in Listing 16-8 is written to support the examples in this part of the book and isn't required when using the standard approach of storing Identity data using Entity Framework Core.

## Creating the User Store

The key interface to implement is `IUserStore<T>`, where `T` is the user class. This interface defines the core features of a user store using the methods described in Table 16-3. All the methods defined by the interface receive a `CancellationToken` parameter, which is used to receive notifications that an asynchronous operation should be canceled, and which is shown as the `token` parameter in the table.

**Table 16-3.** *IUserStore<T> Methods*

Name	Description
<code>CreateAsync(user, token)</code>	This method creates the specified user in the store.
<code>DeleteAsync(user, token)</code>	This method removes the specified user in the store.
<code>UpdateAsync(user, token)</code>	This method updates the specified user in the store.
<code>FindByIdAsync(id, token)</code>	This method retrieves the user with the specified ID from the store.
<code>FindByNameAsync(name, token)</code>	This method retrieves the user with the specified normalized username.
<code>GetUserIdAsync(user, name)</code>	This method returns the ID from the specified user object.
<code>GetUserNameAsync(name, token)</code>	This method returns the username from the specified user object.
<code>SetUserNameAsync(user, name, token)</code>	This method sets the username for the specified user.
<code>GetNormalizedUserNameAsync(user, token)</code>	This method gets the normalized username for the specified user.
<code>SetNormalizedUserNameAsync(user, name, token)</code>	This method sets the normalized username for the specified user.
<code>Dispose()</code>	This method is inherited from the <code>IDisposable</code> interface and is called to release unmanaged resources before the store object is destroyed.

The methods defined by the `IUserStore<T>` interface fall into three groups: core storage (creating/deleting/updating users), querying (locating users by name and ID), and handling names (getting and setting natural and normalized usernames). In the sections that follow, I create a user store by focusing on each group of methods in turn and using the `C#` partial class feature to build up the functionality in multiple class files.

## Implementing the Data Storage Methods

I am going to create a memory-based user store for simplicity. The drawback of this approach, of course, is that changes to the user store will be lost when ASP.NET Core is restarted, but it is enough to get started. Create the `ExampleApp/Identity/Store` folder and add to it a class file named `UserStoreCore.cs`, with the code shown in Listing 16-9.



---

■ **Note** Your code editor may warn you that the `AppUserStore` class doesn't implement all the methods required by the `IUserStore<AppUser>` interface. The code in Listing 16-9 defines a partial class, which means the class members are defined in multiple class files. I will implement the missing methods in the sections that follow.

---

**Listing 16-9.** The Contents of the `UserStoreCore.cs` File in the `Identity/Store` Folder

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

namespace ExampleApp.Identity.Store {

    public partial class UserStore : IUserStore<AppUser> {
        private ConcurrentDictionary<string, AppUser> users
            = new ConcurrentDictionary<string, AppUser>();

        public Task<IdentityResult> CreateAsync(AppUser user,
            CancellationToken token) {
            if (!users.ContainsKey(user.Id) && users.TryAdd(user.Id, user)) {
                return Task.FromResult(IdentityResult.Success);
            }
            return Task.FromResult(Error);
        }

        public Task<IdentityResult> DeleteAsync(AppUser user,
            CancellationToken token) {
            if (users.ContainsKey(user.Id)
                && users.TryRemove(user.Id, out user)) {
                return Task.FromResult(IdentityResult.Success);
            }
            return Task.FromResult(Error);
        }

        public Task<IdentityResult> UpdateAsync(AppUser user,
            CancellationToken token) {
            if (users.ContainsKey(user.Id)) {
                users[user.Id].UpdateFrom(user);
                return Task.FromResult(IdentityResult.Success);
            }
            return Task.FromResult(Error);
        }

        public void Dispose() {
            // do nothing
        }
    }
}
```

```

        private IdentityResult Error => IdentityResult.Failed(new IdentityError {
            Code = "StorageFailure",
            Description = "User Store Error"
        });
    }
}

```

The data structure for the user data is a concurrent dictionary, with each `AppUser` object stored using its `Id` value as the key. Implementing the `CreateAsync`, `DeleteAsync`, and `UpdateAsync` methods means managing the data in the dictionary and producing `IdentityResult` objects to report on the outcome. Successful operations are reported using the `IdentityResult.Success` property.

```

...
return Task.FromResult(IdentityResult.Success);
...

```

Failed operations are reported using the `IdentityResult.Failed` method, which accepts one or more `IdentityError` objects that describe the problem.

```

...
private IdentityResult Error => IdentityResult.Failed(new IdentityError {
    Code = "StorageFailure", Description = "User Store Error"});
...

```

The `IdentityError` class defines `Code` and `Description` properties that are used to describe an error condition. A real user store would be descriptive about the problems it encounters, but for my simple implementation, I produce a general error to indicate a problem.

## Implementing the Search Methods

The next group of methods allows the user store to be searched. Add a class file named `UserStoreQuery.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 16-10.

**Listing 16-10.** The Contents of the `UserStoreQuery.cs` File in the `Identity/Store` Folder

```

using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace ExampleApp.Identity.Store {

    public partial class UserStore {

        public Task<AppUser> FindByIdAsync(string userId,
            CancellationToken token) =>
            Task.FromResult(users.ContainsKey(userId)
                ? users[userId].Clone() : null);

        public Task<AppUser> FindByNameAsync(string normalizedUserName,
            CancellationToken token) =>

```

```

        Task.FromResult(users.Values.FirstOrDefault(user =>
            user.NormalizedUserName == normalizedUserName)?.Clone());
    }
}

```

These methods retrieve `AppUser` objects from the dictionary. In the case of the `FindByIdAsync` method, the `AppUser` objects are stored using `Id` values as keys, which makes the query simple. The `FindByNameAsync` requires more work because the query is performed using the `NormalizedUserName` property, which is not a key. For this method, I use the LINQ `FirstOrDefault` method to locate a matching object.

In both cases, the `Clone` extension method defined in Listing 16-8 is used to create copies of the `AppUser` objects retrieved from the store. This means that any changes made to the `AppUser` objects are not added to the store until the `UpdateAsync` method is called.

## Implementing the ID and Name Methods

The next set of methods is used to get user IDs and get and set usernames. Add a class file named `UserStoreNames.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 16-11.

**Listing 16-11.** The Contents of the `UserStoreNames.cs` File in the `Identity/Store` Folder

```

using System.Threading;
using System.Threading.Tasks;

namespace ExampleApp.Identity.Store {
    public partial class UserStore {
        public Task<string> GetNormalizedUserNameAsync(AppUser user,
            CancellationToken token)
            => Task.FromResult(user.NormalizedUserName);

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

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

        public Task SetNormalizedUserNameAsync(AppUser user,
            string normalizedName, CancellationToken token)
            => Task.FromResult(user.NormalizedUserName = normalizedName);

        public Task SetUserNameAsync(AppUser user, string userName,
            CancellationToken token)
            => Task.FromResult(user.UserName = userName);
    }
}

```

These methods are easy to implement and are mapped directly onto the properties of the `AppUser` class.

## Creating the Normalizer and Seeding the User Store

Normalization of a name is the process of transforming it so queries will match in all the forms in which the name can be expressed. Without normalization, a name such as Alice will be treated differently from alice, ALICE, and AliCE. This can be a problem for data stores, where a query for alice won't match the stored value Alice, for example.

Rather than write complex query matchers, normalization transforms names so the same value is produced regardless of variations in how the name is expressed. For usernames, conventional normalization means converting all the letters to uppercase or lowercase so that all forms of Alice are expressed as alice. ASP.NET Core Identity normalization is done through the `ILookupNormalizer` interface, which defines the methods described in Table 16-4.

**Table 16-4.** *The Methods Defined by the `ILookupNormalizer` Method*

Name	Description
<code>NormalizeName(name)</code>	This method is responsible for normalizing usernames.
<code>NormalizeEmail(email)</code>	This method is responsible for normalizing email addresses.

Creating a custom normalizer isn't required to create a custom user store, but I want to demonstrate the key ASP.NET Core Identity building blocks. Add a class named `Normalizer.cs` to the `ExampleApp/Identity/Store` folder and use it to define the class shown in Listing 16-12.

**Listing 16-12.** The Contents of the `Normalizer.cs` File in the `Identity/Store` Folder

```
using Microsoft.AspNetCore.Identity;

namespace ExampleApp.Identity.Store {

    public class Normalizer : ILookupNormalizer {

        public string NormalizeName(string name)
            => name.Normalize().ToLowerInvariant();

        public string NormalizeEmail(string email)
            => email.Normalize().ToLowerInvariant();
    }
}
```

It doesn't matter how values are normalized if the process is consistent, and all the ways that a name can be expressed are addressed. It is also a good idea to take advantage of the .NET Unicode normalization feature, which ensures that complex characters are handled consistently. In Listing 16-12, I call the `string.Normalize` method and transform the result to lowercase.

To complete the user store, add a class file named `UserStore.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 16-13.

**Listing 16-13.** The Contents of the UserStore.cs File in the Identity/Store Folder

```
using Microsoft.AspNetCore.Identity;

namespace ExampleApp.Identity.Store {

    public partial class UserStore {

        public ILookupNormalizer Normalizer { get; set; }

        public UserStore(ILookupNormalizer normalizer) {
            Normalizer = normalizer;
            SeedStore();
        }

        private void SeedStore() {

            int idCounter = 0;

            foreach (string name in UsersAndClaims.Users) {
                AppUser user = new AppUser {
                    Id = (++idCounter).ToString(),
                    UserName = name,
                    NormalizedUserName = Normalizer.NormalizeName(name)
                };
                users.TryAdd(user.Id, user);
            }
        }
    }
}
```

This code adds a constructor that accepts a `ILookupNormalizer` parameter. ASP.NET Core Identity finds the functionality it requires using ASP.NET Core services. The user store will be set up as a service, and when the ASP.NET Core dependency injection feature instantiates the `UserStore` class, it will instantiate the `ILookupNormalizer` service to provide the constructor argument.

The constructor assigns the normalizer to a property and calls the `SeedStore` method, which populates the user store with `AppUser` objects based on the usernames defined in the `UsersAndClaims` class.

## Configuring Identity and the Custom Services

The application must be configured to set up the custom user store and ASP.NET Core Identity, as shown in Listing 16-14.

**Listing 16-14.** Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;
using Microsoft.AspNetCore.Authentication.Cookies;
```

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using ExampleApp.Identity;
using ExampleApp.Identity.Store;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {

            //services.AddTransient<IAuthorizationHandler,
            //    CustomRequirementHandler>();

            services.AddSingleton<ILookupNormalizer, Normalizer>();
            services.AddSingleton<IUserStore<AppUser>, UserStore>();

            services.AddIdentityCore<AppUser>();

            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");
            });
        }
    }
}

```

The custom implementations of the Identity interfaces are registered as services using the `AddSingleton` method, and Identity is added to the application with the `AddIdentityCore<T>` method, where `T` is the user class. Identity will use its default implementation for any service that has not already been registered, which means you can be selective about the customization you make.

---

■ **Note** The `AddIdentityCore<T>` method is useful when you want greater control over the Identity services and features that are enabled, but the methods used in Part 1 are more suitable for most projects.

---

## Accessing the User Store

Applications don't interface with the user store directly. Instead, Identity provides the `UserManager<T>` class, where `T` is the user class. The `UserManager<T>` class defines a lot of members, and Table 16-5 describes those that relate to the user store. I will describe additional members as I start using other Identity features.

**Table 16-5.** *Selected UserManager<T> Members*

Name	Description
<code>FindByIdAsync(id)</code>	This method locates an <code>AppUser</code> object user by ID by calling the store's <code>FindByIdAsync</code> method.
<code>FindByNameAsync(username)</code>	This method locates an <code>AppUser</code> object user by username. The <code>username</code> argument is normalized and passed to the store's <code>FindByNameAsync</code> method.
<code>CreateAsync(user)</code>	This method adds the specified <code>AppUser</code> object to the store. The security stamp is set, the user is subjected to validation, and the normalized name and email properties are updated, after which the user object is passed to the store's <code>CreateAsync</code> method. (Storing email addresses is described in the “Adding Optional Store Features” section, and validating a user is described in the “Validating User Data” section.)
<code>UpdateAsync(user)</code>	This method applies the user store's update sequence, described in the following sidebar, committing any changes that have been made.
<code>DeleteAsync(user)</code>	This method removes an <code>AppUser</code> object from the store by passing the user object to the store's <code>DeleteAsync</code> method.
<code>GetUserIdAsync(user)</code>	This method gets the ID for the user object by calling the store's <code>GetUserIdAsync</code> method.
<code>GetUserNameAsync(user)</code>	This method gets the name for the user object by calling the store's <code>GetUserNameAsync</code> method.
<code>SetUserNameAsync(user, name)</code>	This method sets the name for the user object by calling the store's <code>SetUserNameAsync</code> method, after which the security stamp is updated and the user manager's update sequence is performed. The update sequence is described in the following sidebar.

The most important `UserManager<T>` methods are `CreateAsync` and `UpdateAsync`. These methods validate the user object (as described in the “Validating User Data” section), ensure that normalized properties are updated consistently, and create new security stamps (which I describe in Chapter 17).

The `userManager<T>` methods that update individual properties are optional, and you can choose to use them or work directly with the properties defined by the user class. The advantage of the `userManager<T>` methods is they often perform useful additional work that I describe as I introduce each set of methods.

Setting properties directly works well with the ASP.NET Core model binding feature, making it easy to work with HTML forms, even though this means you must take care to explicitly perform the additional work that `userManager<T>` does automatically. You must also remember to call the `UpdateAsync` method to apply changes to the user store.

In most cases, I work directly with user class properties in this part of the book because of the close fit with model binding, which suits the style of the examples. You don't have to follow this approach in your own projects, and I include details of the work performed by each `userManager<T>` method I describe.

## UNDERSTANDING THE USER MANAGER UPDATE SEQUENCE

As I introduce the features provided by the `userManager<T>`, many of the descriptions of the methods will refer to the user manager update sequence. The `userManager<T>` class defines a protected method that is called by many other methods to update data in the store. This method goes through a sequence of steps to prepare a user object. This sequence depends on features that I introduce in later chapters, but for now, it is enough to have a rough sense of the process.

This update sequence performs validation (described in the “Validating User Data” section) and updates the normalized name and email properties (I describe storing email addresses in the “Adding Support for Storing Email Addresses” section) before passing the object to the user store's `UpdateAsync` method.

## Working with User Store Data

To prepare for working with the `userManager<T>` class, add the expressions shown in Listing 16-15 to the `_ViewImports.cshtml` file in the Pages folder.

**Listing 16-15.** Adding Expressions in the `_ViewImports.cshtml` File in the Pages Folder

```
@namespace ExampleApp.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.AspNetCore.Identity
@using System.Security.Claims
@using ExampleApp.Identity
```

Defining these namespaces in a view imports file means that I don't have to import them in each of the Razor Pages that uses Identity features.

Next, create the `Pages/Store` folder and add to it a Razor Page named `Users.cshtml` with the content shown in Listing 16-16.



**Listing 16-16.** The Contents of the Users.cshtml File in the Pages/Store Folder

```

@page "/users/{searchname?}"
@model ExampleApp.Pages.Store.FindUserModel

<div class="m-2">
    <form method="get" class="mb-2" action="/users">
        <div class="container-fluid">
            <div class="row">
                <div class="col-9">
                    <input name="searchname" class="w-100" value="@Model.Searchname"
                        placeholder="Enter Username or ID" />
                </div>
                <div class="col-auto">
                    <button type="submit"
                        class="btn btn-primary btn-sm">Find</button>
                    <a class="btn btn-secondary btn-sm" href="/users">Clear</a>
                </div>
            </div>
        </div>
    </form>
    @if (Model.Users?.Count() > 0) {
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr><th>Username</th><th>Normalized</th><th/></tr>
            </thead>
            <tbody>
                @foreach (AppUser user in Model.Users) {
                    <tr>
                        <td>@user.UserName</td>
                        <td>@user.NormalizedUserName</td>
                        <td>
                            <form asp-page-handler="delete" method="post">
                                <partial name="_UserTableRow" model="@user.Id" />
                                <input type="hidden" name="id" value="@user.Id" />
                                <button type="submit" class="btn btn-sm btn-danger">
                                    Delete
                                </button>
                            </form>
                        </td>
                    </tr>
                }
            </tbody>
        </table>
    } else if (!string.IsNullOrEmpty(Model.Searchname)) {
        <h6>No match</h6>
    }
    <a asp-page="edituser" class="btn btn-primary">Create</a>
</div>

```

The Razor Page presents an input element that will allow the user to enter a search term and a table that will display details of the users returned by the search. Each row in the table will display a summary of the user and a series of buttons that will be used to manage the data in the store.

To create the page model, the `Users.cshtml.cs` class file to define the page model class shown in Listing 16-17. If you created the `Users.cshtml` file using the Visual Studio Razor Page template, the class file will already have been created. If you are using Visual Studio Code, add a file named `Users.cshtml.cs` to the `ExampleApp/Pages/Store` folder.

**Listing 16-17.** The Contents of the `Users.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 FindUserModel : PageModel {

        public FindUserModel(UserManager<AppUser> userManager) {
            UserManager = userManager;
        }

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

        public IEnumerable<AppUser> Users { get; set; }
            = Enumerable.Empty<AppUser>();

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

        public async Task OnGet() {
            if (Searchname != null) {
                AppUser nameUser = await UserManager.FindByNameAsync(Searchname);
                if (nameUser != null) {
                    Users = Users.Append(nameUser);
                }
                AppUser idUser = await UserManager.FindByIdAsync(Searchname);
                if (idUser != null) {
                    Users = Users.Append(idUser);
                }
            }
        }

        public async Task<IActionResult> OnPostDelete(string id) {
            AppUser user = await UserManager.FindByIdAsync(id);
            if (user != null) {
                await UserManager.DeleteAsync(user);
            }
        }
    }
}
```

```

        return RedirectToPage();
    }
}

```

The page model class for this Razor Page declares a dependency on the `userManager<AppUser>` service, through which it searches the store for users, either by name or by ID. The POST handler method receives an ID value and uses it to delete a user by calling the `userManager<T>.DeleteAsync` method.

To create the partial view that will display the buttons that will lead to other management features, add a Razor View named `_UserTableRow.cshtml` to the `Pages/Store` folder with the content shown in Listing 16-18.

**Listing 16-18.** The Contents of the `_UserTableRow.cshtml` File in the `Pages/Store` Folder

```

@model string

<a asp-page="edituser" asp-route-id="@Model" class="btn btn-sm btn-secondary">
    Edit
</a>

```

When the user clicks the Edit button, the browser is redirected to a Razor Page named `EditUser`, with a route variable named `id` that provides the ID of the selected user. Add a Razor Page named `EditUser.cshtml` to the `Pages/Store` folder with the content shown in Listing 16-19.

**Listing 16-19.** The Contents of 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" />
            </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 Razor Page displays a table that allows user object properties to be displayed and edited, along with a Save button that will apply changes in the user store and a Cancel button that will return to the Users Razor Page. I am going to build up the editing support gradually as I introduce Identity features, and each feature will have its own partial view that adds rows to the table.

Use the `EditUser.cshtml.cs` file to define the page model class shown in Listing 16-20. You will have to create this file if you are using Visual Studio Code.

**Listing 16-20.** The Contents of the `EditUser.cshtml.cs` File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class UsersModel : PageModel {

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

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

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

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

        public async Task<IActionResult> OnPost(AppUser user) {
            IdentityResult result;
            AppUser storeUser = await UserManager.FindByIdAsync(user.Id);
            if (storeUser == null) {
                result = await UserManager.CreateAsync(user);
            } else {
                storeUser.UpdateFrom(user);
                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 GET page handler receives an ID value, which is used to locate the user object. The POST handler method relies on the ASP.NET Core model binder to create an `AppUser` object from the HTTP request. If there isn't an object in the store with the ID received in the request, the object is stored with the `CreateAsync` method. If there is an existing object, then I copy the property values from the `AppUser` object created by the model binder to the object retrieved from the user store and then use the `UpdateAsync` method to store the changes.

```
...
storeUser.UpdateFrom(user);
result = await UserManager.UpdateAsync(storeUser);
...
```

This approach allows me to deal with HTTP requests that provide values for only some of the properties defined by the user class. Without this step, I would overwrite the data in the store with `null` or default values for those properties for which no value is available in the HTTP request.

The `EditUser` Razor Page relies on a partial view to display the individual fields for editing, which will make it easier for me to add features later. Add a Razor View named `_EditUserBasic.cshtml` to the `Pages/Store` folder, with the content shown in Listing 16-21.

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

```
@model AppUser
<tr>
    <td>ID</td>
    <td>@Model.Id</td>
</tr>
<tr>
    <td>Username</td>
    <td>
        <input class="w-00" asp-for="UserName" />
    </td>
</tr>
<tr>
    <td>Normalized UserName</td>
    <td>
        @(Model.NormalizedUserName ?? "(Not Set)")
        <input type="hidden" asp-for="NormalizedUserName" />
    </td>
</tr>
```

Restart ASP.NET Core, request `http://localhost:5000/users`, enter 1 into the text field and click the Find button. Click the Edit button, change the username to AliceSmith, and click the Save button. The updated summary shows that the new username has been stored, as shown in Figure 16-2.

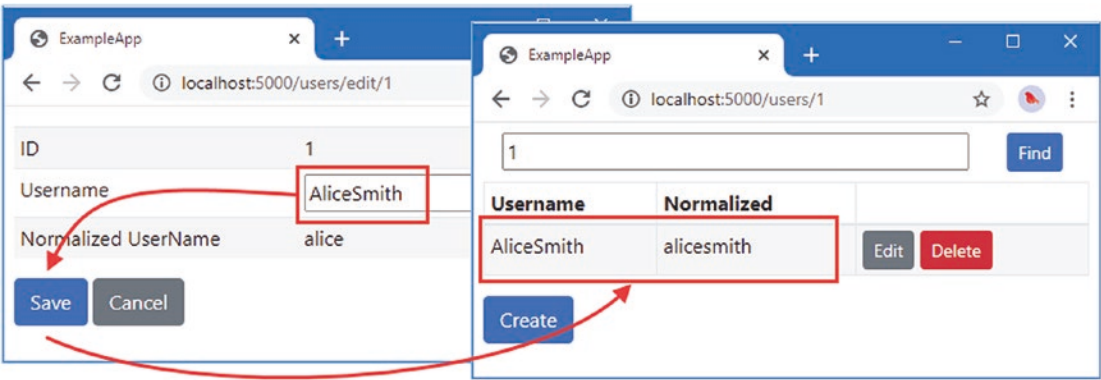


Figure 16-2. Accessing user data

The results of operations on the store are described using `IdentityResult` objects, which I described earlier in the chapter. When an operation fails, I add details of the problem to the model state so that I can use the ASP.NET Core data validation features to present errors to the user. You can see how this works later in the chapter.

Notice that the normalized username is updated when you change the `UserName` property. This is part of the process performed by the `CreateAsync` and `UpdateAsync` methods described in Table 16-3.

## Adding Optional Store Features

The user store works, but it doesn't contain enough data to be useful. Identity uses a series of optional interfaces that stores implement to declare they can store additional data types. Table 16-6 describes the most important interfaces. (There are additional interfaces that support signing in and authentication, which I describe in later chapters.)

Table 16-6. Optional User Store Interfaces

Name	Description
<code>IQueryableUserStore&lt;T&gt;</code>	This interface is implemented by user stores that allow users to be queried using LINQ. I implement this interface in the “Querying the User Store” section.
<code>IUserEmailStore&lt;T&gt;</code>	This interface is implemented by user stores that can manage email addresses. I implement this interface later in this section.
<code>IUserPhoneNumberStore&lt;T&gt;</code>	This interface is implemented by user stores that can manage phone numbers and addresses. I implement this interface later in this section.
<code>IUserPasswordStore&lt;T&gt;</code>	This interface is implemented by user stores that can manage passwords. I use this interface in Chapter 18.
<code>IUserClaimStore&lt;T&gt;</code>	This interface is implemented by user stores that can manage claims. I use this interface in Chapter 17.
<code>IUserRoleStore&lt;T&gt;</code>	This interface is implemented by user stores that can manage roles. I use this interface in Chapter 17.

# Adding Support for Querying the User Store

The Razor Page created in the previous section lets you look up a user by name or ID but requires an exact match. A more flexible approach is to allow the user data to be queried with LINQ and user stores that can be queried implement the `IQueryableUserStore<T>` interface, which defines the property described in Table 16-7.

**Table 16-7.** *The IQueryableUserStore<T> Interface*

Name	Description
Users	This property returns an <code>IQueryable&lt;T&gt;</code> object, where T is the store's user class.

The approach I have taken for the custom user store in this chapter makes it easy to add new features by adding another partial class. To add support for querying user data, add a class file named `UserStoreQueryable.cs` to the `Identity/Store` folder, with the code shown in Listing 16-22.

**Listing 16-22.** The Contents of the UserStoreQueryable.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {

    public partial class UserStore : IQueryableUserStore<AppUser> {

        public IQueryable<AppUser> Users => users.Values
            .Select(user => user.Clone()).AsQueryable<AppUser>();
    }
}
```

To implement the interface, I use the `AsQueryable<T>` extension method, which LINQ provides for converting `IEnumerable<T>` objects into `IQueryable<T>` objects so they can be used in LINQ queries. The LINQ `Select` method is used to duplicate the `AppUser` objects so that changes are not added to the store until an explicit update is performed.

# Querying the User Store

The `UserManager<T>` class defines two properties that are used to access the functionality provided through the `IQueryableStore<T>` interface, as described in Table 16-8.

**Table 16-8.** *The UserManager<T> Properties for Queryable User Stores*

Name	Description
<code>SupportsQueryableUsers</code>	This property returns true if the user store implements the <code>IQueryableUserStore&lt;T&gt;</code> interface.
Users	This property returns an <code>IQueryable&lt;T&gt;</code> object, where T is the store's user class.

In Listing 16-23, I have updated the GET handler in the Users Razor Page model class to use the properties in Table 16-8.

**Listing 16-23.** Querying Users in the Users.cshtml.cs File in the Pages/Store Folder

```
...
public async Task OnGet() {
    if (UserManager.SupportsQueryableUsers) {
        string normalizedName =
            UserManager.NormalizeName(Searchname ?? string.Empty);
        Users = string.IsNullOrEmpty(Searchname)
            ? UserManager.Users.OrderBy(u => u.UserName)
            : UserManager.Users.Where(user => user.Id == Searchname ||
                user.NormalizedUserName.Contains(normalizedName))
                .OrderBy(u => u.UserName);
    } else if (Searchname != null) {
        AppUser nameUser = await UserManager.FindByNameAsync(Searchname);
        if (nameUser != null) {
            Users = Users.Append(nameUser);
        }
        AppUser idUser = await UserManager.FindByIdAsync(Searchname);
        if (idUser != null) {
            Users = Users.Append(idUser);
        }
    }
}
...
```

It is important to check the value of the SupportsQueryableUsers property before performing a query because reading the Users property will cause an exception if the store doesn't implement the IQueryableStore<T> interface.

In the listing, I use the UserManager<T>.Users property with the LINQ Where method to find AppUser object's whose NormalizedUserName property contains the search term entered into the text field or whose Id property matches the search term exactly. All users are shown if there is no search term specified.

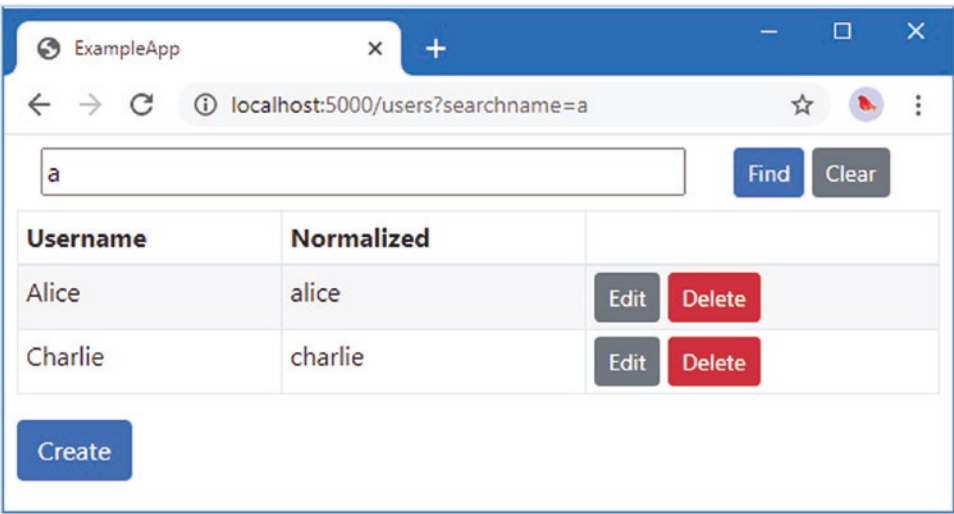
I normalized the search term to perform the LINQ query, which I can do using the normalization methods provided by the UserManager<T> class, described in Table 16-9. These are convenience methods so that components don't have to declare dependencies directly on the ILookupNormalizer service.

**Table 16-9.** UserManager<T> Normalization Convenience Methods

Name	Description
NormalizeName(name)	This method invokes the NormalizeName method on the ILookupNormalizer service.
NormalizeEmail(email)	This method invokes the NormalizeEmail method on the ILookupNormalizer service.

To perform a query on the user store, restart ASP.NET Core, and request `http://localhost:5000/users`. Enter a into the text field and click the Find button. Two matches will be displayed, as shown in Figure 16-3.





**Figure 16-3.** Querying the user store

## Adding Support for Storing Email Addresses and Phone Numbers

Most applications store email addresses and phone numbers, either to identify users or to communicate with them. User stores that can manage email addresses implement the `IUserEmailStore<T>` interface, where `T` is the user class. The `IUserEmailStore<T>` interface defines the methods shown in Table 16-10. (As with other user store interfaces, all of the methods shown in the table define a token parameter through which a `CancellationToken` object is provided, allowing notifications to be received when asynchronous operations are canceled.)

**Table 16-10.** The `IUserEmailStore<T>` Methods

Name	Description
<code>FindByEmailAsync(email, token)</code>	This method returns the user class instance that has the specified normalized email address.
<code>GetEmailAsync(user, token)</code>	This method returns the email address of the specified user object.
<code>SetEmailAsync(user, email, token)</code>	This method sets the email address for the specified user object.
<code>GetNormalizedEmailAsync(user, token)</code>	This method returns the normalized email address of the specified user object.
<code>SetNormalizedEmailAsync(user, email, token)</code>	This method sets the normalized email address for the specified user object.
<code>GetEmailConfirmedAsync(user, token)</code>	This method gets the value of the property used to indicate whether the email address has been confirmed, which I demonstrate in Chapter 17.
<code>SetEmailConfirmedAsync(user, confirmed, token)</code>	This method sets the value of the property used to indicate whether the email address has been confirmed, which I demonstrate in Chapter 17.

User stores that can store phone numbers implement the `IUserPhoneNumberStore<T>` interface, which defines the methods described in Table 16-11.

**Table 16-11.** *The IUserPhoneNumberStore<T> Methods*

Name	Description
<code>SetPhoneNumberAsync(user, phone, token)</code>	This method sets the telephone number for the specified user.
<code>GetPhoneNumberAsync(user, token)</code>	This method sets the telephone number for the specified user.
<code>GetPhoneNumberConfirmedAsync(user, token)</code>	This method gets the value of the property used to indicate whether the phone number has been confirmed, which I demonstrate in Chapter 17.
<code>SetPhoneNumberConfirmedAsync(user, token)</code>	This method sets the value of the property used to indicate whether the phone number has been confirmed, which I demonstrate in Chapter 17.

The first step is to add properties to the user class to store the email and phone data, as shown in Listing 16-24.

**Listing 16-24.** Adding Properties to the User Class in the `AppUser.cs` File in the Identity Folder

```
using System;

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; }
    }
}
```

As with the username, Identity stores regular and normalized versions of the email address, so I have added two properties to the user class for these values. I have also added properties for confirming the user's email address and phone number, which I describe in Chapter 17.

Next, add a class file named `UserStoreEmail.cs` to the `Identity/Store` method and use it to define the partial class shown in Listing 16-25, which extends the user store class to support the `IUserEmailStore<T>` interface.

**Listing 16-25.** The Contents of the UserStoreEmail.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {

    public partial class UserStore : IUserEmailStore<AppUser> {

        public Task<AppUser> FindByEmailAsync(string normalizedEmail,
            CancellationToken token) =>
            Task.FromResult(Users.FirstOrDefault(user =>
                user.NormalizedEmailAddress == normalizedEmail));

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

        public Task SetEmailAsync(AppUser user, string email,
            CancellationToken token) {
            user.EmailAddress = email;
            return Task.CompletedTask;
        }

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

        public Task SetNormalizedEmailAsync(AppUser user, string normalizedEmail,
            CancellationToken token) {
            user.NormalizedEmailAddress = normalizedEmail;
            return Task.CompletedTask;
        }

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

        public Task SetEmailConfirmedAsync(AppUser user, bool confirmed,
            CancellationToken token) {
            user.EmailAddressConfirmed = confirmed;
            return Task.CompletedTask;
        }
    }
}
```

Adding support for optional interfaces is simple once you have the core features in place. The `FindByEmailAsync` method is implemented using a LINQ query against the `Users` property defined by the `IQueryableUserStore<T>` interface. The `GetEmailAsync`, `SetEmailAsync`, `GetNormalizedEmailAsync`, and `SetNormalizedEmailAsync` methods rely on the properties added to the user class in Listing 16-25. Chapter 17 describes the `GetEmailConfirmedAsync` and `SetEmailConfirmedAsync` methods.

Next, add a class file named `UserStorePhone.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 16-26, which extends the user store to implement the `IUserPhoneNumberStore<T>` interface.

**Listing 16-26.** The Contents of the `UserStorePhone.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 : IUserPhoneNumberStore<AppUser> {
        public Task<string> GetPhoneNumberAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.PhoneNumber);

        public Task SetPhoneNumberAsync(AppUser user, string phoneNumber,
            CancellationToken token) {
            user.PhoneNumber = phoneNumber;
            return Task.CompletedTask;
        }

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

        public Task SetPhoneNumberConfirmedAsync(AppUser user, bool confirmed,
            CancellationToken token) {
            user.PhoneNumberConfirmed = confirmed;
            return Task.CompletedTask;
        }
    }
}
```

To seed the user store with email addresses and phone numbers, add the statements shown in Listing 16-27 to the `UserStore.cs` file in the `Identity/Store` folder.

**Listing 16-27.** Adding Seed Data in the `UserStore.cs` File in the `Identity/Store` Folder

```
using Microsoft.AspNetCore.Identity;

namespace ExampleApp.Identity.Store {

    public partial class UserStore {

        public ILookupNormalizer Normalizer { get; set; }
```

```

public UserStore(ILookupNormalizer normalizer) {
    Normalizer = normalizer;
    SeedStore();
}

private void SeedStore() {

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

```

The changes in the listing generate email addresses in the `example.com` domain for each user and assign every user the same phone number. The email addresses are normalized using the `NormalizeEmail` method described in Table 16-4.

## Using Email Addresses and Phone Numbers

The `UserManager<T>` class defines the members shown in Table 16-12 for dealing with email addresses.

**Table 16-12.** *The `UserManager<T>` Members for Email Addresses*

Name	Description
<code>SupportsUserEmail</code>	This property returns true if the store implements the <code>IUserEmailStore&lt;T&gt;</code> interface.
<code>FindByEmailAsync(email)</code>	This method locates a user by email address. The email address is normalized before it is passed to the store's <code>FindByEmailAsync</code> method.
<code>GetEmailAsync(user)</code>	This method returns the email address for the specified instance of the user class.
<code>SetEmailAsync(user, email)</code>	This method sets the email address for the specified instance of the user class. The email address is passed to the store's <code>SetEmailAsync</code> method, and the <code>SetEmailConfirmedAsync</code> method is called to set the confirmed state to false, after which the security stamp is updated and the user store's update sequence is applied. Chapter 17 describes security stamps.

To allow the email address to be edited, add a Razor View named `_EditUserEmail.cshtml` to the `Pages/Store` folder with the content shown in Listing 16-28.

**Listing 16-28.** The Contents of the `_EditUserEmail.cshtml` File in the `Pages/Store` Folder

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

@if (UserManager.SupportsUserEmail) {
    <tr>
        <td>Email</td>
        <td>
            <input class="w-00" asp-for="EmailAddress" />
        </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>
}
```

There is also a set of members for managing phone numbers, as described in Table 16-13.

**Table 16-13.** The `UserManager<T>` Members for Phone Numbers

Name	Description
<code>SupportsUserPhoneNumber</code>	This property returns true if the store implements the <code>IUserPhoneNumberStore&lt;T&gt;</code> interface.
<code>GetPhoneNumberAsync(user)</code>	This method gets the phone number for the specified user by calling the store's <code>GetPhoneNumberAsync</code> method.
<code>SetPhoneNumberAsync(user, number)</code>	This method sets the phone number for the user by calling the store's <code>SetPhoneNumberAsync</code> method. The store's <code>SetPhoneNumberConfirmedAsync</code> method is called to set the confirmed state to false, the security stamp is updated (as explained in Chapter 17), the user is subject to validation (as described in the "Validating User Data" section), and the user manager's update sequence is applied.

To allow the phone number to be set, create a Razor View named `_EditUserPhone.cshtml` in the `Pages/Store` folder with the content shown in Listing 16-29.

**Listing 16-29.** The Contents of the `_EditUserPhone.cshtml` File in the Pages/Store Folder

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

@if (UserManager.SupportsUserPhoneNumber) {
    <tr>
        <td>Phone</td>
        <td>
            <input class="w-00" asp-for="PhoneNumber" />
            <input type="hidden" asp-for="PhoneNumberConfirmed" />
        </td>
    </tr>
}
```

Add the elements shown in Listing 16-30 to the `EditUser.cshtml` file in the Pages/Store folder to incorporate the new views into the application.

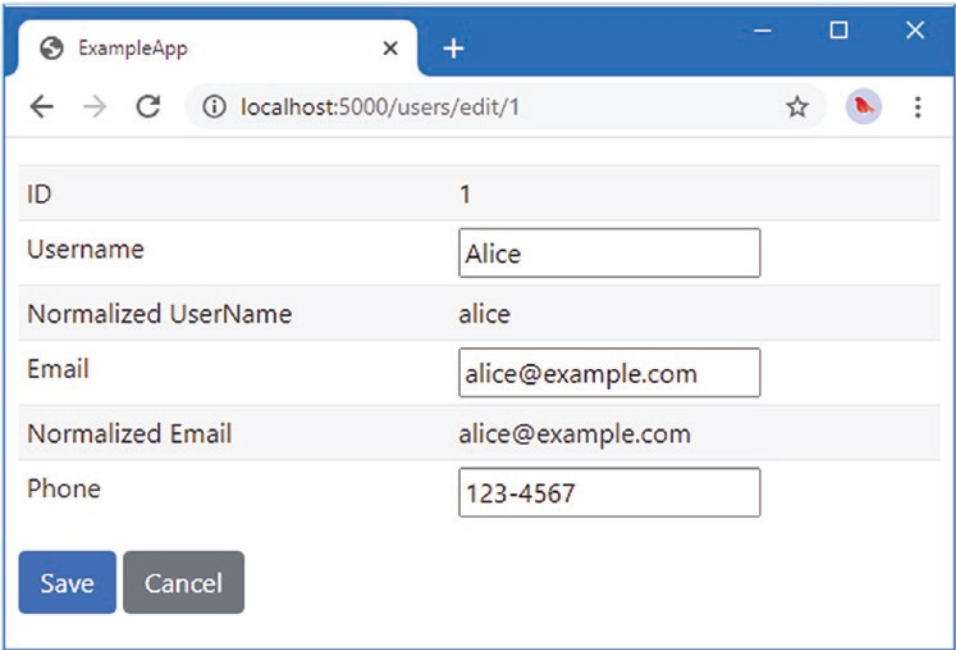
**Listing 16-30.** Adding Partial Views in the `EditUser.cshtml` File in the Pages/Store Folder

```
@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" />
            </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 to see the additional properties added for email addresses and phone numbers, as shown in Figure 16-4.



**Figure 16-4.** Adding support for storing email addresses and phone numbers

## Adding Custom User Class Properties

User stores can define additional properties that are specific to an application. This is a feature that pre-dates the use of claims and isn't needed because any piece of data about the user can be represented with a claim. But it is still supported, so Listing 16-31 adds new properties to the `AppUser` class. (I explain how to manage claims in the user store in Chapter 17.)

---

■ **Note** Custom user class properties are not as useful as they appear because they are not added to the data provider to ASP.NET Core when a user signs in unless you also create a custom translation class, as demonstrated in Chapter 18. My advice is to store all additional data as claims, which is simpler and will automatically be presented to ASP.NET Core.

---

**Listing 16-31.** Adding Custom Properties in the `AppUser.cs` File in the Identity Folder

```
using System;

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; }
}
}

```

The FavoriteFood and Hobby properties are string properties. Add a Razor View named `_EditUserCustom.cshtml` to the Pages/Store folder and use it to define the partial view shown in Listing 16-32, which contains elements required to edit the properties added to the user class in Listing 31.

**Listing 16-32.** The Contents of the `_EditUserCustom.cshtml` File in the Pages/Store Folder

```

@model AppUser

<tr>
    <td>Favorite Food</td>
    <td><input class="w-100" asp-for="FavoriteFood" /></td>
</tr>
<tr>
    <td>Hobby</td>
    <td><input class="w-100" asp-for="Hobby" /></td>
</tr>

```

Add the element shown in Listing 16-33 to the `EditUser.cshtml` file in the Pages/Store folder to incorporate the partial view into the application.

**Listing 16-33.** Adding an Element 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" />
            </tbody>
        </table>
    </form>

```

```

        <div>
            <button type="submit" class="btn btn-primary">Save</button>
            <a asp-page="users" class="btn btn-secondary">Cancel</a>
        </div>
    </form>
</div>

```

To seed the user store with values for the new properties, add the statements shown in Listing 16-34 to the `UserStore.cs` file in the `Identity/Store` folder.

**Listing 16-34.** Seeding the Store in the `UserStore.cs` File in the `Identity/Store` Folder

```

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

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
                };
            }
        }
    }
}

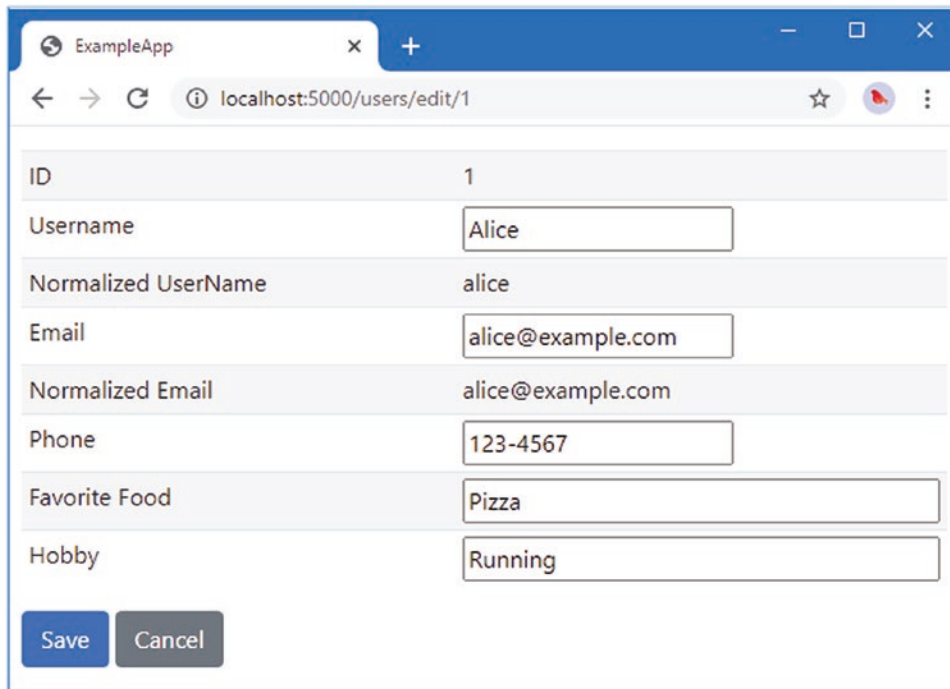
```

```

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

```

Restart ASP.NET Core, request <http://localhost:5000/users>, and click one of the Edit buttons. You will see the custom properties, populated with the seed data, as shown in Figure 16-5.



ID	1
Username	<input type="text" value="Alice"/>
Normalized UserName	alice
Email	<input type="text" value="alice@example.com"/>
Normalized Email	alice@example.com
Phone	<input type="text" value="123-4567"/>
Favorite Food	<input type="text" value="Pizza"/>
Hobby	<input type="text" value="Running"/>

**Figure 16-5.** Adding custom properties to the user class

## Validating User Data

Before they pass user objects to the store, the `CreateAsync` and `UpdateAsync` methods validate the user object. To see an example of validation, request <http://localhost:5000/users>, click the Edit button for Bob, and enter Charlie into the `UserName` field. Click Save, and you will see the response shown in Figure 16-6.

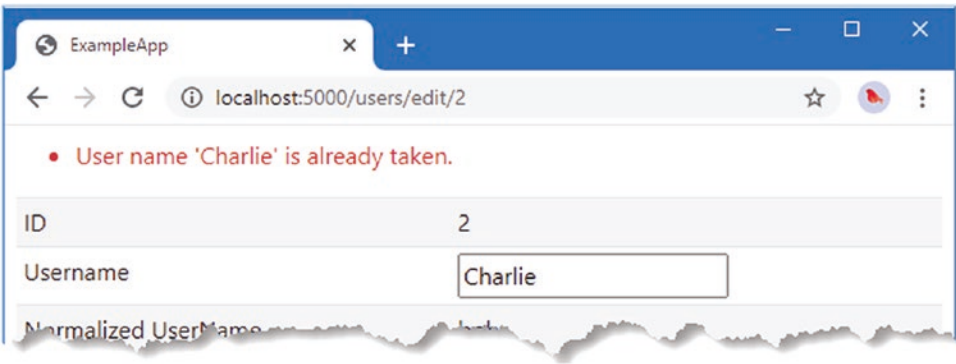


Figure 16-6. A validation response for user data

Validation constraints are expressed using the `IValidator<T>` interface, where `T` is the user class. The `userManager<T>` class uses dependency injection to receive the set of `IValidator<T>` services that have been registered and validate user objects using the method described in Table 16-14.

Table 16-14. The `IValidator<T>` Method

Name	Description
<code>ValidateAsync(manager, user)</code>	This method validates the specified user object and can access the user store through the <code>userManager&lt;T&gt;</code> parameter. The validation result is expressed with an <code>IdentityResult</code> object.

The default implementation of the `IValidator<T>` interface checks that usernames and email addresses are unique and contain only permitted characters. It is this validator that produced the error message shown in Figure 16-6.

CONFIGURING THE DEFAULT USER VALIDATOR

The default validator can be configured using the options pattern in the `Startup` class. Two configuration options are available, accessed through the `IdentityOptions.User` property, like this:

```
...
services.AddIdentityCore<AppUser>(opts => {
    opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
    opts.User.RequireUniqueEmail = true;
});
...
```

The `AllowedUserNameCharacters` property specifies the set of characters that usernames can contain. The default value allows uppercase and lowercase characters, numbers, and some symbols. The `RequireUniqueEmail` property specifies whether email addresses must be unique and defaults to `false`.

To create a custom validator, add a class file named `EmailValidator.cs` to the `ExampleApp/Identity` folder with the code shown in Listing 16-35.

**Listing 16-35.** The Contents of the `EmailValidator.cs` File in the Identity Folder

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

namespace ExampleApp.Identity {
    public class EmailValidator : IUserValidator<AppUser> {
        private static string[] AllowedDomains = new[] { "example.com", "acme.com" };
        private static IdentityError err
            = new IdentityError { Description = "Email address domain not allowed" };

        public EmailValidator(ILookupNormalizer normalizer) {
            Normalizer = normalizer;
        }

        private ILookupNormalizer Normalizer { get; set; }

        public Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
            AppUser user) {
            string normalizedEmail = Normalizer.NormalizeEmail(user.EmailAddress);
            if (AllowedDomains.Any(domain =>
                normalizedEmail.EndsWith($"@{domain}"))) {
                return Task.FromResult(IdentityResult.Success);
            }
            return Task.FromResult(IdentityResult.Failed(err));
        }
    }
}
```

Care must be taken when registering custom validators. Identity will only add the default validator if there are no existing `IUserValidator<T>` services before the `AddIdentityCore` method is called. This means you should register custom services before the `AddIdentityCore` method if you want to replace the default validator and after the `AddIdentityCore` method if you want to retain the default validator. I want to supplement the default validation, so I registered my custom class after setting up Identity, as shown in Listing 16-36.

**Listing 16-36.** Registering a User Validator in the `Startup.cs` File in the `ExampleApp` Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILookupNormalizer, Normalizer>();
    services.AddSingleton<IUserStore<AppUser>, UserStore>();

    services.AddIdentityCore<AppUser>();

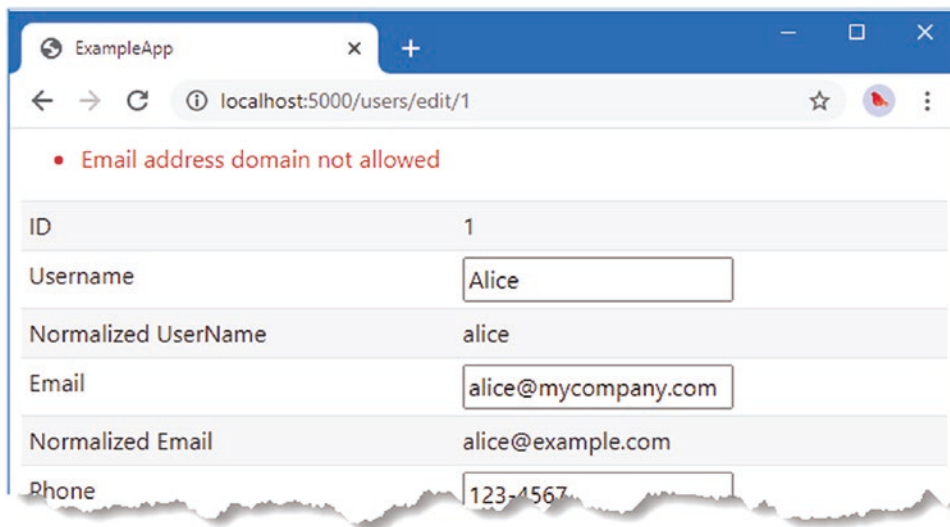
    services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();
}
```

```

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();
}
...

```

To see the effect of the new validator, restart ASP.NET Core, request `http://localhost:5000/users`, and click the Edit button for the Alice user. Change the Email field to `alice@mycompany.com` and click the Save button. The new validator class will produce the error shown in Figure 16-7.



**Figure 16-7.** A custom user validator

## Summary

In this chapter, I described the interfaces that are implemented by user store. I created a custom user store that supports core features, as well as support for LINQ queries, email addresses and phone numbers, and custom user class properties. I also explained how user data can be validated before it is added to the store. In the next chapter, I add support for claims, roles, and user confirmations.