

CHAPTER 19



Creating a Role Store

In this chapter, I create and use a role store. As earlier chapters have shown, Identity doesn't need a role store to use roles, and the use of a role store is entirely optional and the features that are provided are not needed in most projects. That said, a role store can be a useful way to ensure that roles are used consistently. Table 19-1 puts role stores in context.

Table 19-1. Putting Roles Stores in Context

Question	Answer
What is it?	The role store allows additional data to be associated with roles.
Why is it useful?	The role store has some useful validation features and can be used to store additional claims, which are added to the ClaimsPrincipal objects created when a user signs in.
How is it used?	A role class and an implementation of the IRoleStore<T> interface are created.
Are there any pitfalls or limitations?	Most projects do not need the features that the role store provides, which is not widely understood.
Are there any alternatives?	The role store is an optional feature and is not required to use roles in ASP.NET Core Identity.

Table 19-2 summarizes the chapter.

Table 19-2. Chapter Summary

Problem	Solution	Listing
Create a role store	Define a role class and create an implementation of the IRoleStore<T> interface. Register the implementation as a service for dependency injection.	2-5, 7,8
Support LINQ queries in a custom role store	Implement the IQueryableRoleStore<T> interface.	6
Manage the roles in the role store	Use the RoleManager<T> class.	9, 10, 20-23
Validate roles before they are added to the role store	Create an implementation of the IRoleValidator<T> interface.	11, 12
Enforce role consistency	Use the role store as the master list of permissible role memberships	13, 14
Store claims with roles	Implement the IRoleClaimStore<T> interface.	15-19

Preparing for This Chapter

This chapter uses the ExampleApp project from Chapter 18. 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 19-1 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 19-1. 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 19-1. The data is stored only in memory, and changes will be lost when ASP.NET Core is stopped.

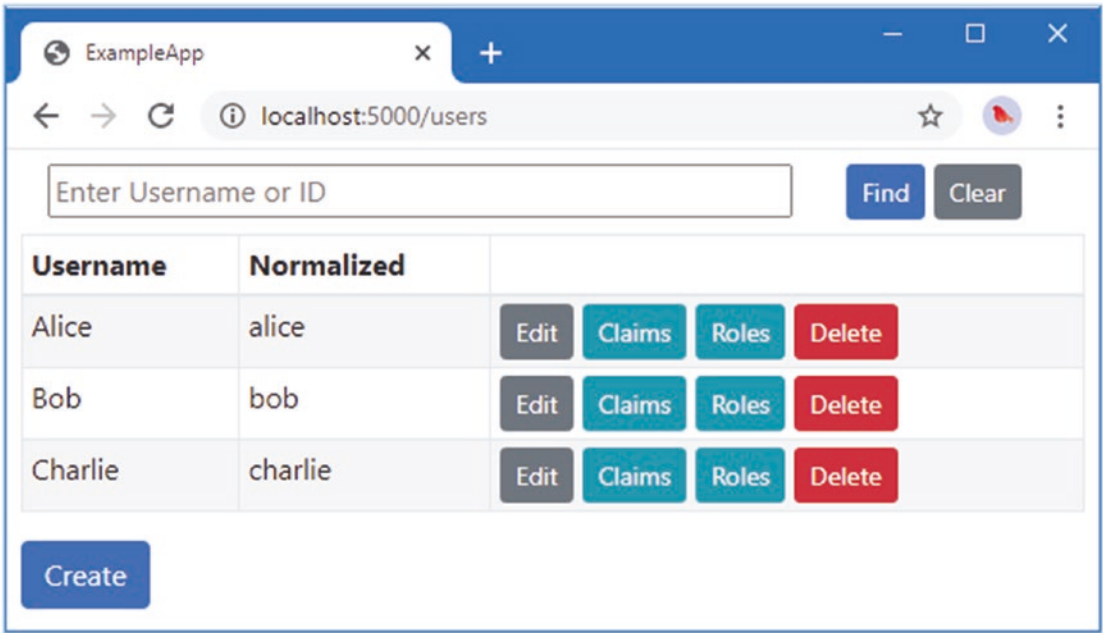


Figure 19-1. Running the example application

Creating a Custom Role Store

At the moment, roles are represented in the user store as claims. If an authorization policy requires role membership, access will be granted to users who have a `Role` claim with the required role name. This approach is suitable for most web applications, which typically rely on a small number of roles, but there are some limitations.

The first limitation is that a role doesn't exist until the first time a user is granted a `Role` claim with a specific value. So, for example, there is no `Manager` role known to the application until the first time a `Role` claim with a value of `Manager` is created for a user. This can lead to quirks in the application because it is difficult to determine if a specific role exists but there are no users assigned to it or if that role doesn't exist at all. All the application can do is query the claims in the user store.

The second limitation is that it is difficult to enforce consistency, which means that typos can cause unexpected application behaviors. It is easy to mistype role names so that some users are assigned to the `Administrator` role (singular) while others are assigned to the `Administrators` role (plural). Users will either be denied access to features to which they are entitled or, worse, be granted access to features from which they should be excluded.

To help manage complex sets of roles, Identity supports a *role store*, which is used to define and manage the roles an application requires. In the sections that follow, I create a role store for the example application and demonstrate its use.

Creating the Role Class

The starting point is to define a class that will be instantiated to represent roles. This class, known as the *role class*, plays the same role as the user class does in the user store. Add a class file named `AppRole.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 19-2.

Listing 19-2. The Contents of the `AppRole.cs` File in the Identity Folder

```
using System;

namespace ExampleApp.Identity {

    public class AppRole {

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

        public string Name { get; set; }

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

There are no restrictions on how the role class is defined, and I started with a minimum setup of an `Id` property that will uniquely identify a role, a `Name` property that will be displayed to users, and a `NormalizedName`, which will be used to store a normalized representation of the `Name` value and will help ensure that comparisons using names are consistent.

Creating the Role Store

Role stores are defined by the `IRoleStore<T>` interface, where `T` is the role class. I am going to follow the same approach I took for the user store and build up the role store gradually using partial classes, each of which implements a group of related methods from the `IRoleStore<T>` interface. Table 19-3 describes the methods defined by the `IRoleStore<T>` interface. The token parameter defined by the methods in Table 19-3 is a `CancellationToken` object that is used to receive notifications when an asynchronous task is canceled.

Table 19-3. *The `IRoleStore<T>` Methods*

Name	Description
<code>CreateAsync(role, token)</code>	This method adds the specified role to the store.
<code>UpdateAsync(role, token)</code>	This method updates the specified role in the store, committing any pending changes persistently.
<code>DeleteAsync(role, token)</code>	This method removes the specified role object from the role store.
<code>GetRoleIdAsync(role, token)</code>	This method gets the ID of the specified role object.
<code>GetRoleNameAsync(role, token)</code>	This method gets the name of the specified role object.
<code>SetRoleNameAsync(role, name, token)</code>	This method sets the name of the specified role object.
<code>GetNormalizedRoleNameAsync(role, token)</code>	This method gets the normalized name of the specified role object.
<code>SetNormalizedRoleNameAsync(role, name, token)</code>	This method sets the normalized name of the specified role object.
<code>FindByIdAsync(id, token)</code>	This method retrieves the role object with the specified ID from the store.
<code>FindByNameAsync(name, token)</code>	This method retrieves the role object with the specified normalized name from the store.
<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.

All the methods described in Table 19-3 return an `IdentityResult` object that indicates the outcome of the operation and provides details of any problems that occur.

Implementing the Data Storage Methods

The data storage methods defined by the `IRoleStore<T>` interface address managing the collection of roles managed by the store. Add a class file named `RoleStoreCore.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 19-3. (Your code editor will indicate an error for this class until all the partial classes in this part of the chapter have been created.)

Listing 19-3. The Contents of the RoleStoreCore.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 RoleStore: IRoleStore<AppRole> {
        private ConcurrentDictionary<string, AppRole> roles
            = new ConcurrentDictionary<string, AppRole>();

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

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

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

        public void Dispose() {
            // do nothing
        }

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

```

I have followed the same approach for storing roles as I did for user objects in Chapter 16. The data will be stored in memory using a `ConcurrentDictionary` object, with the `AppRole.Id` property used as the key. Restarting ASP.NET Core will reset the role store, and any changes are lost.

Implementing the Name Methods

The next set of methods is responsible for providing Identity with access to the basic information about the role, which is done by mapping the methods onto the properties defined by the role object. Add a class file named `RoleStoreNames.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 19-4.

Listing 19-4. The Contents of the `RoleStoreNames.cs` File in the `Identity/Store` Folder

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

namespace ExampleApp.Identity.Store {

    public partial class RoleStore {

        public Task<string> GetRoleIdAsync(AppRole role, CancellationToken token)
            => Task.FromResult(role.Id);

        public Task<string> GetRoleNameAsync(AppRole role, CancellationToken token)
            => Task.FromResult(role.Name);

        public Task SetRoleNameAsync(AppRole role, string roleName,
            CancellationToken token) {
            role.Name = roleName;
            return Task.CompletedTask;
        }

        public Task<string> GetNormalizedRoleNameAsync(AppRole role,
            CancellationToken token) => Task.FromResult(role.NormalizedName);

        public Task SetNormalizedRoleNameAsync(AppRole role, string normalizedName,
            CancellationToken token) {
            role.NormalizedName = normalizedName;
            return Task.CompletedTask;
        }
    }
}
```

Implementing the Search Methods

The next set of methods allow role objects to be located by ID or name. Add a class file named `RoleStoreQuery.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 19-5.

Listing 19-5. The Contents of the `RoleStoreQuery.cs` File in the `Identity/Store` Folder

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

```
namespace ExampleApp.Identity.Store {

    public partial class RoleStore {

        public Task<AppRole> FindByIdAsync(string id, CancellationToken token)
            => Task.FromResult(roles.ContainsKey(id) ? roles[id].Clone() : null);

        public Task<AppRole> FindByNameAsync(string name, CancellationToken token)
            => Task.FromResult(roles.Values.FirstOrDefault(r => r.NormalizedName ==
                name)?.Clone());

    }

}
```

Notice that these methods use the `Clone` extension method defined in Listing 19-5 to create objects that can be modified by the application and then committed to the store. This is an important measure that prevents changes that the user abandons from being stored.

Making the Store Queryable

Role stores can implement the optional `IQueryableRoleStore<T>` interface, where `T` is the role class, to create a role store that can be queried easily with LINQ. The `IQueryableRoleStore<T>` defines the property described in Table 19-4.

Table 19-4. The `IQueryableRoleStore<T>` Interface

Name	Description
Roles	This property returns an <code>IQueryable<T></code> object, where <code>T</code> is the role class.

To implement the optional interface, make the changes shown in Listing 19-6 to the partial class defined in the `RoleStoreQuery.cs` file.

Listing 19-6. Implementing an Interface in the `RoleStoreQuery.cs` File in the `Identity/Store` Folder

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

namespace ExampleApp.Identity.Store {

    public partial class RoleStore: IQueryableRoleStore<AppRole> {

        public Task<AppRole> FindByIdAsync(string id, CancellationToken token)
            => Task.FromResult(roles.ContainsKey(id) ? roles[id].Clone() : null);

        public Task<AppRole> FindByNameAsync(string name, CancellationToken token)
            => Task.FromResult(roles.Values.FirstOrDefault(r => r.NormalizedName ==
                name)?.Clone());

    }
```

```

        public IQueryable<AppRole> Roles =>
            roles.Values.Select(role => role.Clone()).AsQueryable<AppRole>();
    }
}

```

Once again, it is important to return objects that can be modified outside of the store, such that changes are stored only when the `UpdateAsync` method is called.

Seeding the Role Store and Configuring the Application

Now I have implemented the `IRoleStore<T>` and `IQueryableRoleStore<T>` interfaces, I can finish the store by adding some seed data. Add a class file named `RoleStore.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 19-7.

Listing 19-7. The Contents of the `RoleStore.cs` File in the `Identity/Store` Folder

```

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

namespace ExampleApp.Identity.Store {

    public partial class RoleStore {

        public ILookupNormalizer Normalizer { get; set; }

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

        private void SeedStore() {

            var roleData = new List<string> {
                "Administrator", "User", "Sales", "Support"
            };

            int idCounter = 0;

            foreach (string roleName in roleData) {
                AppRole role = new AppRole {
                    Id = (++idCounter).ToString(),
                    Name = roleName,
                    NormalizedName = Normalizer.NormalizeName(roleName)
                };
                roles.TryAdd(role.Id, role);
            }
        }
    }
}

```


The code in Listing 19-7 seeds the role store with four roles. Role names are normalized using an implementation of the `ILookupNormalizer` interface, which I introduced in Chapter 16.

Listing 19-8 configures the application to use the role store.

Listing 19-8. Using a Role Store 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<ISMSender, ConsoleSMSSender>();
            //services.AddSingleton<IUserClaimsPrincipalFactory<AppUser>,
            //    AppUserClaimsPrincipalFactory>();
            services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();
            services.AddSingleton<IRoleStore<AppRole>, RoleStore>();

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

                opts.Password.RequireNonAlphanumeric = false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
                opts.Password.RequiredLength = 8;

            })
            .AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
            .AddTokenProvider<PhoneConfirmationTokenGenerator>
                (TokenOptions.DefaultPhoneProvider)
            .AddSignInManager()
            .AddRoles<AppRole>();
        }
    }
}
```

```

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

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

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

    app.UseStaticFiles();
    app.UseAuthentication();
    app.UseRouting();
    app.UseAuthorization();

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

```

The `AddSingleton` method is used to define a service so that the `RoleStore` class is used to resolve dependencies on the `IRoleStore<AppRole>` interface.

The `AddRoles<T>` method is used to specify the class that will represent roles, which is `AppUser` in this case. This is a convenience method that registers a service for the `RoleManager<T>` class (described in the next section) and sets up the default role validator service (described in the “Validating Roles” section).

I have moved the statement that sets up the custom claims principal factory service. The `AddRoles` method sets up a default factory service, even if one is already registered, which is the opposite behavior of most Identity methods. I have moved the statement that registers the `AppUserClaimsPrincipalFactory` class as the factory service so that it replaces the service created by the `AddRoles` method.

■ **Note** Identity provides an `AddRoleStore<T>` extension method that can be used to specify the role store class. Care must be taken when using this method because it creates a scoped service, which means that a new instance of the role store class will be created for every HTTP request. A scoped service is useful when storing data with Entity Framework Core, but for this chapter, I need to explicitly define the service with the `AddSingleton` method so that all requests share a single instance of the `RoleStore` class.

Managing Roles

Roles are managed through the `RoleManager<T>` class, where `T` is the role class. Table 19-5 describes the basic members defined by the `RoleManager<T>` class. There are additional members, which I describe later in the chapter.

Table 19-5. *The Basic RoleManager<T> Members*

Name	Description
<code>SupportsQueryableRoles</code>	This property returns <code>true</code> if the role store implements the <code>IQueryableRoleStore<T></code> interface.
<code>Roles</code>	This property returns an <code>IQueryable<T></code> object when the role store implements the <code>IQueryableRoleStore<T></code> interface. An exception will be thrown if this property is read and the user store doesn't implement the interface.
<code>CreateAsync(role)</code>	This method adds the specified role object to the store. The role is subjected to validation (described in the "Validating Roles" section), and the normalized name is set before being passed to the role store's <code>CreateAsync</code> method.
<code>UpdateAsync(role)</code>	This method updates the specified role in the store, persisting any changes that have been made. The role is subjected to validation (described in the "Validating Roles" section), and the normalized name is set before being passed to the role store's <code>UpdateAsync</code> method.
<code>DeleteAsync(role)</code>	This method removes the specified role from the store by calling the role store's <code>DeleteAsync</code> method.
<code>RoleExistsAsync(name)</code>	This method returns <code>true</code> if the role store contains a role with the specified name, which is normalized before being passed to the role store.
<code>FindByIdAsync(id)</code>	This method returns the role with the specified ID, or <code>null</code> if there is no such role.
<code>FindByNameAsync(name)</code>	This method returns the role with the specified name or <code>null</code> if there is no such role. The role name is normalized before it is passed to the role store's <code>FindByNameAsync</code> method.

To manage the roles in the store, add a Razor Page named `Roles.cshtml` to the `Pages/Store` folder with the content shown in Listing 19-9.

Listing 19-9. The Contents of the `Roles.cshtml` File in the `Pages/Store` Folder

```
@page "/"roles"
@model ExampleApp.Pages.Store.RolesModel

<h4 class="bg-secondary text-white text-center p-2">Roles</h4>

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

<table class="table table-striped table-sm">
  <thead><tr><th>Name</th><th># Users in Role</th></tr></thead>
```

```

<tbody>
  @foreach (AppRole role in Model.Roles) {
    <tr>
      <td class="pl-2">
        <input name="name" form="@role.Id" value="@role.Name" />
      </td>
      <td>@((await Model.GetUsersInRole(role)).Count())</td>
      <td class="text-right pr-2">
        <form method="post" id="@role.Id">
          <input type="hidden" name="id" value="@role.Id" />
          <button type="submit" class="btn btn-danger btn-sm"
            asp-page-handler="delete">Delete</button>
          <button type="submit" class="btn btn-info btn-sm"
            asp-page-handler="save">Save</button>
        </form>
      </td>
    </tr>
  }
  <tr>
    <td>
      <input name="name" form="newRole" placeholder="Enter Role Name" />
    </td>
    <td></td>
    <td class="text-right pr-2">
      <form method="post" id="newRole">
        <button type="submit" class="btn btn-info btn-sm"
          asp-page-handler="create">
            Create
        </button>
      </form>
    </td>
  </tr>
</tbody>
</table>

```

The page displays the roles known to the application in a table, allowing the name to be edited or the role to be deleted. There is also a table row that allows a new role to be created.

Add the code shown in Listing 19-10 to the Roles.cshtml.cs file to define the page model. You will have to create this file if you are using Visual Studio Code.

Listing 19-10. The Contents of the Roles.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 RolesModel : PageModel {

        public RolesModel(UserManager<AppUser> userManager,
            RoleManager<AppRole> roleManager) {
            UserManager = userManager;
            RoleManager = roleManager;
        }

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

        public IEnumerable<AppRole> Roles => RoleManager.Roles.OrderBy(r => r.Name);

        public async Task<IList<AppUser>> GetUsersInRole(AppRole role) =>
            await UserManager.GetUsersInRoleAsync(role.Name);

        public async Task<IActionResult> OnPostDelete(string id) {
            AppRole role = await RoleManager.FindByIdAsync(id);
            if (role != null) {
                IdentityResult result = await RoleManager.DeleteAsync(role);
                if (!result.Succeeded) {
                    return ProcessErrors(result.Errors);
                }
            }
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostSave(AppRole editedRole) {
            IdentityResult result = await RoleManager.UpdateAsync(editedRole);
            if (!result.Succeeded) {
                return ProcessErrors(result.Errors);
            }

            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostCreate(AppRole newRole) {
            IdentityResult result = await RoleManager.CreateAsync(newRole);
            if (!result.Succeeded) {
                return ProcessErrors(result.Errors);
            }
            return RedirectToPage();
        }

        private IActionResult ProcessErrors(IEnumerable<IdentityError> errors) {
            foreach (IdentityError err in errors) {
                ModelState.AddModelError("", err.Description);
            }
        }
    }
}

```

```
        return Page();
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/roles`, and you will see the list of roles from the role store, as shown in Figure 19-2. You can change the name of a role, delete a role, and create a new role.

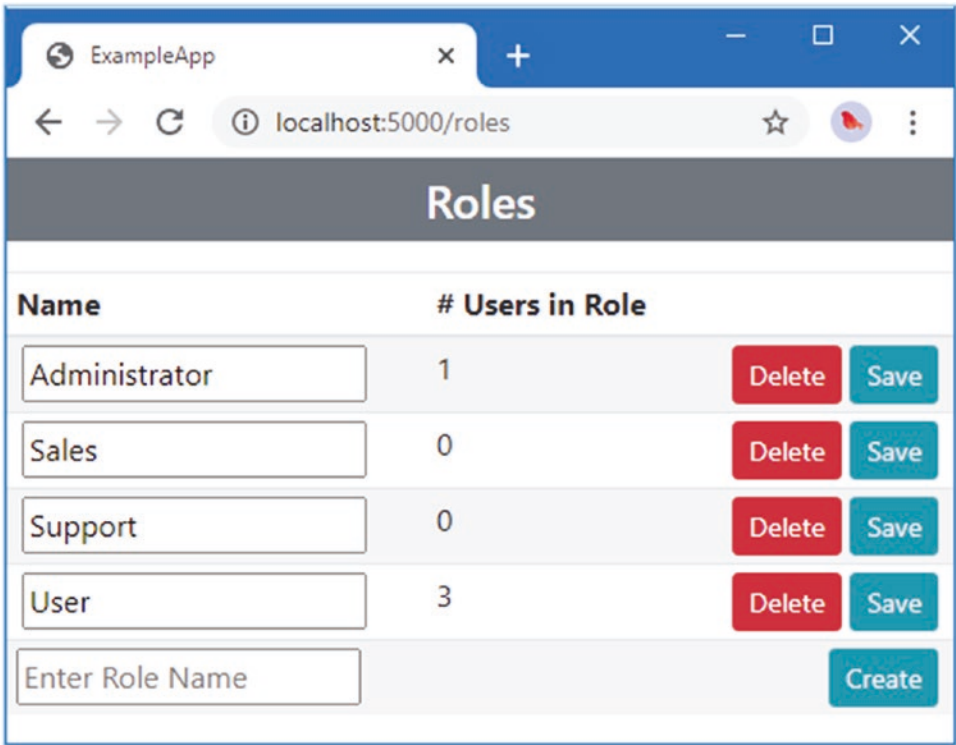


Figure 19-2. Managing roles

It is important to understand that the authoritative repository for role memberships is the user store, not the role store. This can seem counterintuitive, but, as you will see in later examples, the role store exists to supplement the features provided by the user store and not to replace them.

So, for example, I had to use the `userManager<T>` class in Listing 19-10 to determine how many users have been assigned to each role stored in the role store. The roles in the role store may not be the complete set of roles used by the application, which you can see by changing the name of the User role and then changing it back, which produces the sequence shown in Figure 19-3.

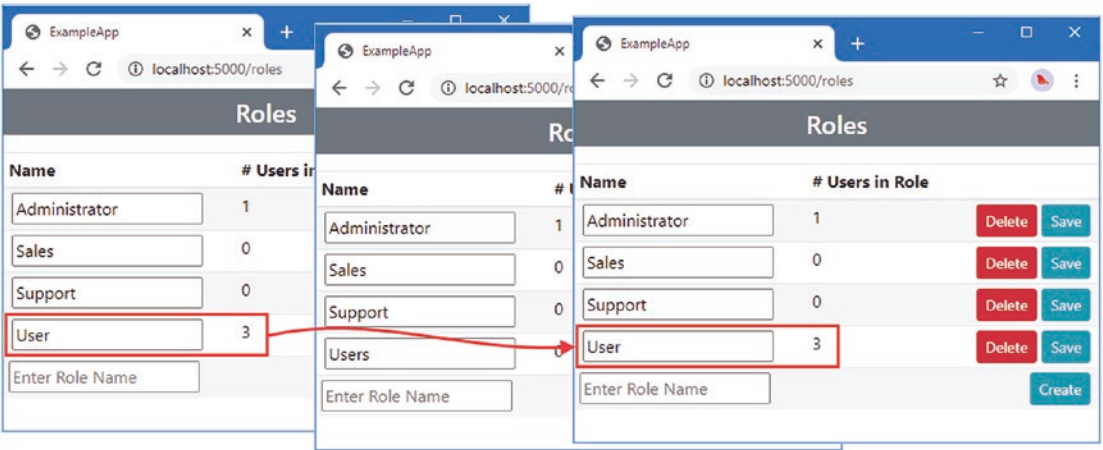


Figure 19-3. Understanding the relationship between the user and role stores

When you rename the role, there is no longer a User role in the role store, but that doesn't stop users from being assigned to that role because that data is in the user store.

Validating Roles

The `RoleManager<T>` class's `CreateAsync` and `UpdateAsync` methods perform a validation check before passing the role onto the role store. Role validation is performed by services that implement the `IRoleValidator<T>` interface, where `T` is the role class. The validation interface defines the method described in Table 19-6.

Table 19-6. The Method Defined by the `IRoleValidator<T>` Interface

Name	Description
<code>ValidateAsync(manager, role)</code>	This method is called to validate the specified role on behalf of the specified role manager. The method returns an <code>IdentityResult</code> object, which indicates whether validation was successful and provides details of the validation errors if it was not.

Identity provides a built-in role validator that ensures that role names are not empty strings and role names are unique within the role store. Validation errors are expressed using `IdentityResult` objects that are returned by the `CreateAsync` and `UpdateAsync` methods when roles fail validation. You can see the errors reported by the built-in validator by requesting `http://localhost:5000/roles` and clicking the Create button without entering text into the adjacent text field, as shown in Figure 19-4. Enter the name of an existing role, such as **User**, and click the Create button to see the error reported when a role name is already in use, also shown in Figure 19-4.

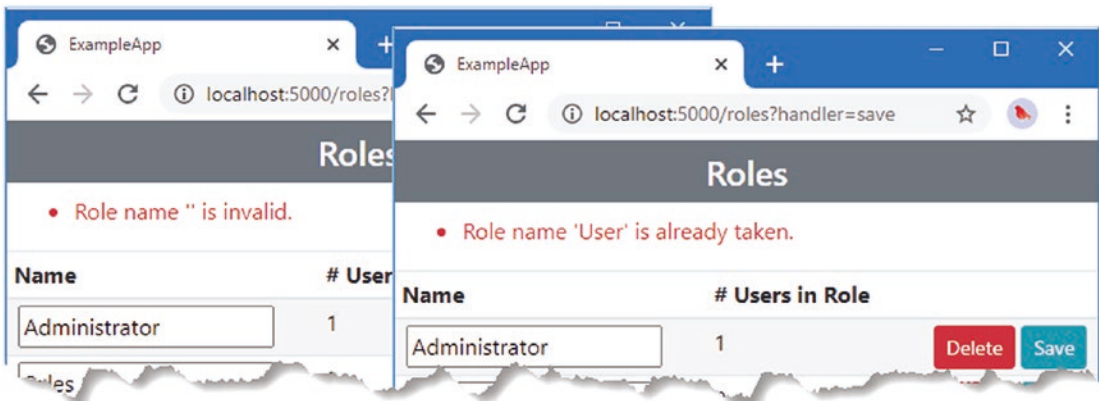


Figure 19-4. The built-in Identity role validation

Creating a Custom Role Validator

To create a custom validator, add a class file named `RoleValidator.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 19-11.

Listing 19-11. The Contents of the `RoleValidator.cs` File in the Identity Folder

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

namespace ExampleApp.Identity {
    public class RoleValidator : IRoleValidator<AppRole> {

        private static IdentityError error = new IdentityError {
            Description = "Names cannot be plural/singular of existing roles"
        };

        public async Task<IdentityResult> ValidateAsync(RoleManager<AppRole> manager,
            AppRole role) {
            if (await manager.FindByNameAsync(role.Name.EndsWith("s")
                ? role.Name[0..^1] : role.Name + "s") == null) {
                return IdentityResult.Success;
            }
            return IdentityResult.Failed(error);
        }
    }
}
```

The validator prevents role names that simply add or omit the letter `s` from the name of an existing role. I find that I start out defining roles with singular names (`administrator`, `user`, etc.) and then accidentally switch to plurals (`administrators`, `users`, and so on). The validator checks for this mistake but still allows roles whose names end with an `s`, such as `Sales`, to be created, just as long as there isn't already a `Sale` role in the store.

In Listing 19-12, I have registered the validator as an implementation of the `IRoleValidator<AppRole>` interface. I have done this after the `AddIdentityCore` method is called so that the new validator is used in addition to the built-in Identity role validator. If I had registered the service before the `AddIdentityCore` method, the custom validator would have replaced the built-in one.

Listing 19-12. Registering the 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.AddSingleton<IEmailSender, ConsoleEmailSender>();
    services.AddSingleton<ISMSender, ConsoleSMSender>();
    //services.AddSingleton<IUserClaimsPrincipalFactory<AppUser>,
    //    AppUserClaimsPrincipalFactory>();
    services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();
    services.AddSingleton<IRoleStore<AppRole>, RoleStore>();

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

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

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

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

```

    services.AddRazorPages();
    services.AddControllersWithViews();
}
...

```

Restart ASP.NET Core and request `http://localhost:5000/roles`. Try to create a role named `Users`, and you will see the error message displayed in Figure 19-5 because there is already a role in the store named `User`.

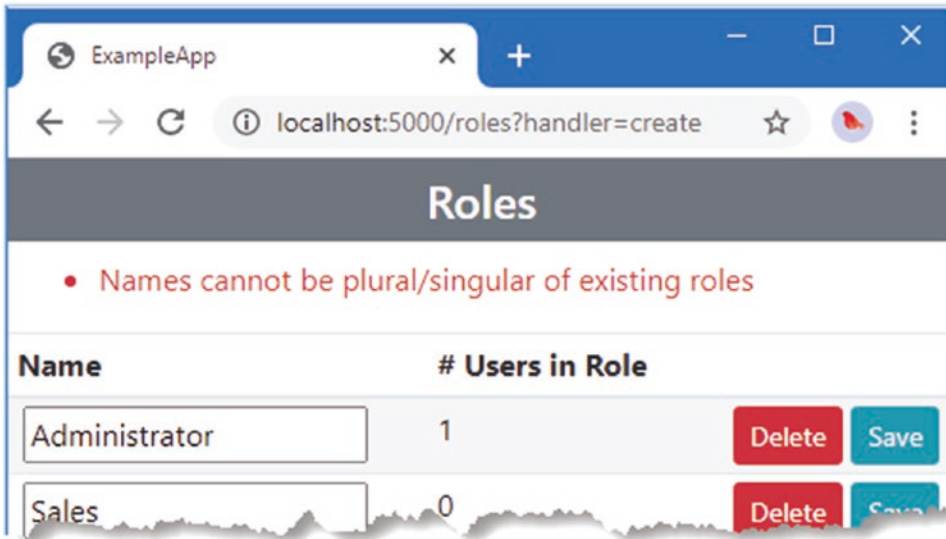


Figure 19-5. Validating roles

Enforcing Role Consistency

One reason to create a role store is to enforce consistency, ensuring that a typo doesn't assign a user to the wrong role. Identity doesn't provide built-in support for restricting roles to names that exist in the role store, but it is a simple process to implement manually. The first step is to restrict the roles to which users can be assigned, as shown in Listing 19-13.

Listing 19-13. Using the Role Store 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;
using Microsoft.AspNetCore.Mvc.Rendering;

```

```

namespace ExampleApp.Pages.Store {

    public class UserRolesModel : PageModel {

        public UserRolesModel(UserManager<AppUser> userManager,
            RoleManager<AppRole> roleManager) {
            UserManager = userManager;
            RoleManager = roleManager;
        }

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

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

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

        public async void OnGet() {
            AppUser user = await GetUser();
            if (user != null) {
                Roles = (await UserManager.GetClaimsAsync(user))
                    .Where(c => c.Type == ClaimTypes.Role).Select(c => c.Value);
                AvailableRoles = new SelectList(RoleManager.Roles
                    .OrderBy(r => r.Name)
                    .Select(r => r.Name).Except(Roles));
            }
        }

        public async Task<IActionResult> OnPostAdd(string 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);
    }
}

```

The changes add a page model property named `AvailableRoles`, which provides a sequence of roles to which the user has not been assigned. Role membership is case-insensitive, but the `UserManager<T>` normalizes role names before they are added to the user store, so I have to compare the `Name` properties of the `AppRole` objects in the role store to populate the `AvailableRoles` property. Listing 19-14 uses a `select` element to constrain the roles that can be chosen.

■ **Note** I have used the `Name`, rather than the `NormalizedName`, properties in Listing 19-13 because of the way I worked around the normalized role name issue described in Chapter 17. If you decide to use normalized role names, then the `NormalizedName` property should be used to enforce role consistency.

Listing 19-14. Constraining Role Selection in 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" />
                        <select asp-items="@Model.AvailableRoles"
                            name="newRole" class="w-100">
                        </select>
                    </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>

```

Restart ASP.NET Core, request <http://localhost:5000/users>, and click the Roles button for Alice. The select element only allows the user to be assigned to roles that are in the store, as shown in Figure 19-6.

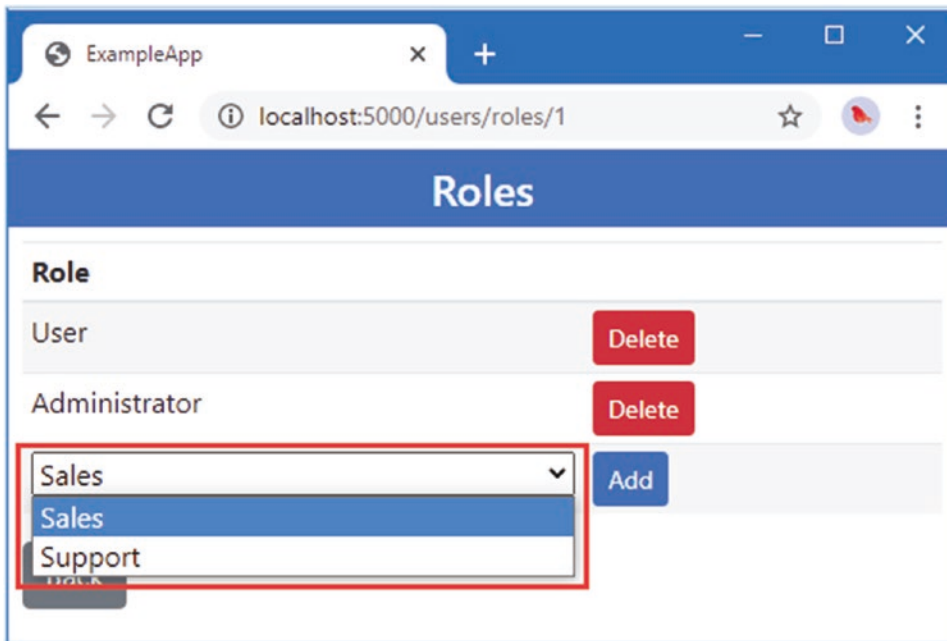


Figure 19-6. Constraining role assignment

Storing Claims with Roles

Roles can be assigned claims so that assigning a user to a role in the store gives the user the claims associated with that role. This is a useful way of managing complex sets of claims consistently and means that you don't have to manually add large sets of claims to individual users. In the sections that follow, I show you how to extend the role store to handle claims and explain how these claims are used.

Extending the Role Class

The first step is to add a property to the role class that will represent the claims associated with a role, as shown in Listing 19-15.

Listing 19-15. Adding a Property in the AppRole.cs File in the Identity Folder

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

namespace ExampleApp.Identity {

    public class AppRole {

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

        public string Name { get; set; }

        public string NormalizedName { get; set; }

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

Extending the Role Store

The `IRoleClaimStore<T>` interface is implemented by role stores that can manage claims, where `T` is the role class. The interface defines the methods described in Table 19-7. (The methods described in the table define a `CancellationToken` parameter. Unlike other interfaces, this parameter is optional.)

Table 19-7. The `IRoleClaimStore<T>` Methods

Name	Description
<code>GetClaimsAsync(role)</code>	This method returns the claims associated with the specified role.
<code>AddClaimAsync(role, claim)</code>	This method adds a claim to the specified role.
<code>RemoveClaimAsync(role, claim)</code>	This method removes a claim from the specified role.

To add claims support to the role store, add a class file named `RoleStoreClaims.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 19-16.

Listing 19-16. The Contents of the RoleStoreClaims.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 RoleStore : IRoleClaimStore<AppRole> {

        public Task AddClaimAsync(AppRole role, Claim claim,
            CancellationToken token = default) {
            role.Claims.Add(claim);
            return Task.CompletedTask;
        }

        public Task<IList<Claim>> GetClaimsAsync(AppRole role,
            CancellationToken token = default) =>
            Task.FromResult(role.Claims ?? new List<Claim>());

        public Task RemoveClaimAsync(AppRole role, Claim claim,
            CancellationToken token = default) {
            role.Claims = role.Claims.Where(c => !(string.Equals(c.Type, claim.Type)
                && string.Equals(c.Value, claim.Value))).ToList<Claim>();
            return Task.CompletedTask;
        }
    }
}

```

The implementation of the interface uses the Claims property added to the AppRole class in Listing 19-16.

Extending the Claims Principal Factory

The claims associated with a role are processed by the claims principal factory. As I explained in Chapter 15, this is the connection point between Identity and the authorization features of ASP.NET Core. In Listing 19-17, I have updated the custom claims principal factory I created in Chapter 18 to get the roles to which the user has been assigned from the user store, retrieve the corresponding role from the role store, and process the role's claims.

Listing 19-17. Processing Role Claims in the AppUserClaimsPrincipalFactory.cs File in the Identity Folder

```

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

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

        public AppUserClaimsPrincipalFactory(UserManager<AppUser> userManager,
            RoleManager<AppRole> roleManager) {
            UserManager = userManager;
            RoleManager = roleManager;
        }

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

```

```

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

    if (UserManager.SupportsUserRole && RoleManager.SupportsRoleClaims) {
        foreach (string roleName in await UserManager.GetRolesAsync(user)) {
            AppRole role = await RoleManager.FindByNameAsync(roleName);
            if (role != null && role.Claims != null) {
                identity.AddClaims(role.Claims);
            }
        }
    }
    return new ClaimsPrincipal(identity);
}
}
}

```

The factory class was set up using the `AddSingleton` method in Listing 19-17, which was fine because the class didn't contain any features that prevented a single instance from processing concurrent requests. The changes in Listing 19-17 included dependencies on the `UserManager<T>` and `RoleManager<T>` services, both of which have a scope lifecycle. Since a singleton service cannot declare a dependency on a scoped service, I have to change the lifecycle of the factory class, as shown in Listing 19-18.

Listing 19-18. Changing a Service Lifecycle in the `Startup.cs` File in the `ExampleApp` Folder

```

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

```

Without this change, an exception will be thrown when ASP.NET Core starts because the dependency injection system won't be able to resolve the dependencies declared by the factory class.

Seeding the Role Store with Claims

To seed the store with claims, add the statements shown in Listing 19-19 to the `RoleStore.cs` file in the `Identity/Store` folder.

Listing 19-19. Seeding Claims in the `RoleStore.cs` File in the `Identity/Store` Folder

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

namespace ExampleApp.Identity.Store {

    public partial class RoleStore {

        public ILookupNormalizer Normalizer { get; set; }

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

        private void SeedStore() {

            var roleData = new List<string> {
                "Administrator", "User", "Sales", "Support"
            };

            var claims = new Dictionary<string, IEnumerable<Claim>> {
                { "Administrator", new [] { new Claim("AccessUserData", "true"),
                    new Claim(ClaimTypes.Role, "Support") } },
                { "Support", new [] { new Claim(ClaimTypes.Role, "User") } } }
            };

            int idCounter = 0;

            foreach (string roleName in roleData) {
                AppRole role = new AppRole {
                    Id = (++idCounter).ToString(),
                    Name = roleName,
                    NormalizedName = Normalizer.NormalizeName(roleName)
                };
                if (claims.ContainsKey(roleName)) {
                    role.Claims = claims[roleName].ToList<Claim>();
                }
                roles.TryAdd(role.Id, role);
            }
        }
    }
}
```

The changes create additional claims so that membership of the Administrator role, for example, will lead to an additional role claim for the Support role and an AccessUserData claim whose value is true (there is no meaning attributed to the AccessUserData claim, which I have created just to show that claims other than roles can be made).

Managing Claims

The `RoleManager<T>` class defines the members shown in Table 19-8 for managing the claims associated with a role.

Table 19-8. *The `RoleManager<T>` Members for Managing Claims*

Name	Description
<code>SupportsRoleClaims</code>	This property returns true if the role store implements the <code>IRoleClaimStore<T></code> interface.
<code>GetClaimsAsync(role)</code>	This method returns the claims associated with the specified role by calling the store's <code>GetClaimsAsync</code> method.
<code>AddClaimAsync(role, claim)</code>	This method adds a claim to the specified role by calling the store's <code>AddClaimAsync</code> method. The role is subjected to validation, and the normalized name is set before being passed to the role store's <code>UpdateAsync</code> method.
<code>RemoveClaimAsync(role, claim)</code>	This method removes a claim from the specified role by calling the store's <code>RemoveClaimAsync</code> method. The role is subjected to validation, and the normalized name is set before being passed to the role store's <code>UpdateAsync</code> method.

To manage the claims associated with a role, add a Razor Page named `RoleClaims.cshtml` to the `Pages/Store` folder with the content shown in Listing 19-20.

Listing 19-20. The Contents of the `RoleClaims.cshtml` File in the `Pages/Store` Folder

```
@page "/roles/claims/{id?}"
@model ExampleApp.Pages.Store.RoleClaimsModel

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

<h4 class="bg-primary text-white text-center p-2">@Model.Role.Name 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.Role.Id, claim, false))" />
                </tr>
            }
        </tbody>
    </table>
</div>
```

```

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

```

The view part of the Razor Page displays a sequence of claims provided by the page model class, using the `_ClaimsRow.cshtml` partial view. To create the page model that will provide the claims, add the code shown in Listing 19-21 to the `RoleClaims.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 19-21. The Contents of the `RoleClaims.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.Security.Claims;
using System.Threading.Tasks;

namespace ExampleApp.Pages.Store {

    public class RoleClaimsModel : PageModel {

        public RoleClaimsModel(RoleManager<AppRole> roleManager)
            => RoleManager = roleManager;

        public RoleManager<AppRole> RoleManager { get; set; }

        public AppRole Role { get; set; }

        public IEnumerable<Claim> Claims => Role.Claims ?? new List<Claim>();

        public async Task OnGet(string id) {
            Role = await RoleManager.FindByIdAsync(id);
        }

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

```

```

        public async Task<IActionResult> OnPostEdit(string id, string type,
            string value, string oldType, string oldValue) {
            Role = await RoleManager.FindByIdAsync(id);
            await RoleManager.RemoveClaimAsync(Role, new Claim(oldType, oldValue));
            await RoleManager.AddClaimAsync(Role, new Claim(type, value));
            return RedirectToPage();
        }

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

```

The page model class defines handler methods that add, edit, and delete claims, using the methods provided by the role manager class described in Table 19-8. To integrate the claim feature into the rest of the application, add the elements shown in Listing 19-22 to the Roles.cshtml file.

Listing 19-22. Integrating Claims into the Roles.cshtml File in the Pages/Store Folder

```

@page "/roles"
@model ExampleApp.Pages.Store.RolesModel

<h4 class="bg-secondary text-white text-center p-2">Roles</h4>

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

<table class="table table-striped table-sm">
    <thead>
        <tr><th>Name</th><th># Users in Role</th><th># Claims</th></tr>
    </thead>
    <tbody>
        @foreach (AppRole role in Model.Roles) {
            <tr>
                <td class="pl-2">
                    <input name="name" form="@role.Id" value="@role.Name" />
                </td>
                <td>@((await Model.GetUsersInRole(role)).Count())</td>
                <td>@((role.Claims?.Count() ?? 0))
                    <a asp-page="RoleClaims" class="btn btn-secondary btn-sm ml-2"
                        asp-route-id="@role.Id">Edit</a>
                </td>
                <td class="text-right pr-2">
                    <form method="post" id="@role.Id">
                        <input type="hidden" name="id" value="@role.Id" />
                        <button type="submit" class="btn btn-danger btn-sm"
                            asp-page-handler="delete">Delete</button>
                    </form>
                </td>
            </tr>
        }
    </tbody>
</table>

```

```

        <button type="submit" class="btn btn-info btn-sm"
            asp-page-handler="save">Save</button>

        </form>
    </td>
</tr>
}
<tr>
    <td>
        <input name="name" form="newRole" placeholder="Enter Role Name" />
    </td>
    <td></td>
    <td></td>
    <td class="text-right pr-2">
        <form method="post" id="newRole">
            <button type="submit" class="btn btn-info btn-sm"
                asp-page-handler="create">
                Create
            </button>
        </form>
    </td>
</tr>
</tbody>
</table>

```

The number of claims for each role is displayed, along with an anchor element that navigates to the RoleClaims Razor Page.

Displaying Claims

To see the way that claims are assigned to the user, add a Razor Page named `LiveClaims.cshtml` to the Pages folder with the content shown in Listing 19-23.

Listing 19-23. The Contents of the `LiveClaims.cshtml` File in the Pages Folder

```

@page
@using System.Security.Claims
@using ExampleApp.Identity
@using Microsoft.AspNetCore.Identity
@inject UserManager<AppUser> UserManager

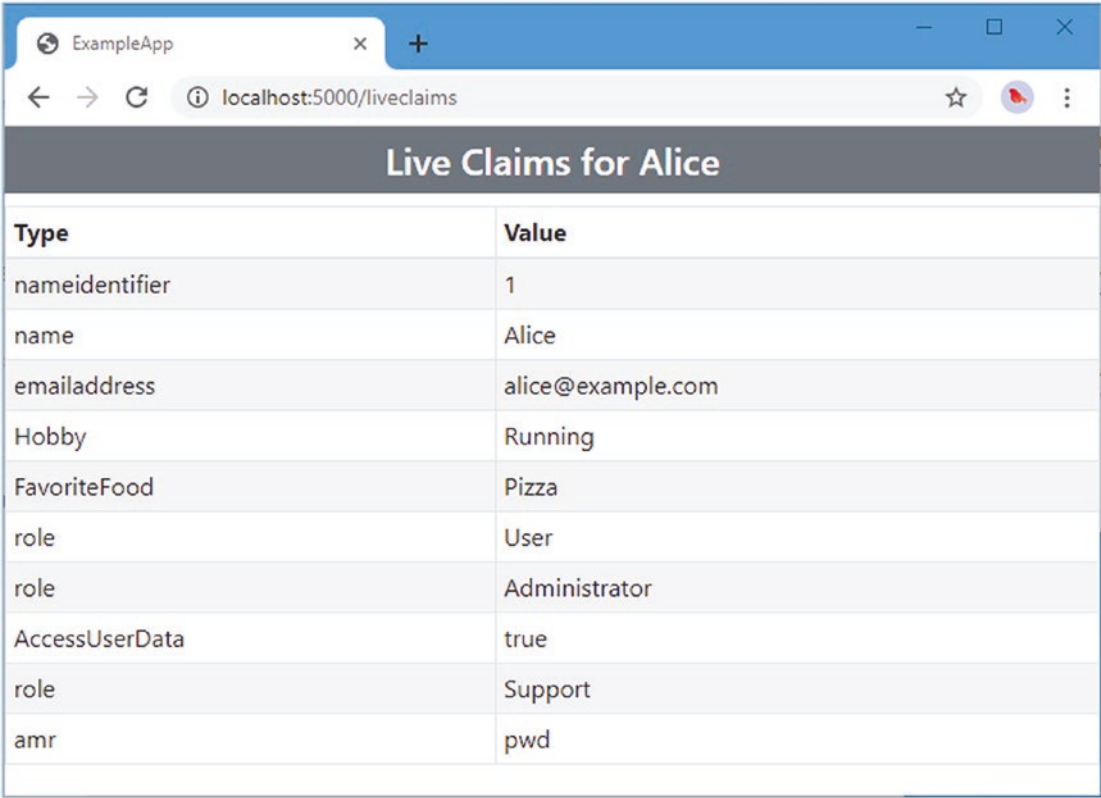
@{
    string GetName(string claimType) =>
        (Uri.IsWellFormedUriString(claimType, UriKind.Absolute)
         ? System.IO.Path.GetFileName(new Uri(claimType).LocalPath)
         : claimType);
}

<h4 class="bg-secondary text-white text-center p-2">
    Live Claims for @(User.Identity.Name ?? "No User")
</h4>

```

```
<table class="table table-sm table-striped table-bordered">
  <thead><tr><th>Type</th><th>Value</th></tr></thead>
  <tbody>
    @foreach (Claim claim in User.Claims) {
      <tr><td>@GetName(claim.Type)</td><td>@claim.Value</td></tr>
    }
  </tbody>
</table>
```

Restart ASP.NET Core, request `http://localhost:5000/signout`, and click the Sign Out button. Select `alice@example.com` from the list, enter **MySecret1\$** into the password field, and click the Sign In button. Navigate to `http://localhost:5000/liveclaims`, and you will see the claims that have been created for the user, as shown in Figure 19-7.

A screenshot of a web browser window titled 'ExampleApp'. The address bar shows 'localhost:5000/liveclaims'. The page has a dark header with the text 'Live Claims for Alice'. Below the header is a table with two columns: 'Type' and 'Value'. The table contains ten rows of claims for a user named Alice.

Type	Value
nameidentifier	1
name	Alice
emailaddress	alice@example.com
Hobby	Running
FavoriteFood	Pizza
role	User
role	Administrator
AccessUserData	true
role	Support
amr	pwd

Figure 19-7. Claims derived from roles

As the figure shows, the set of claims created by the factory incorporates the additional role and the `AccessUserData` claims defined in the seed data.

■ **Note** One of the claims shown in Figure 19-7 has the type `amr` and the value `pwd`. This claim is created by the `SignInManager<T>` class, as I explain in Chapter 20.

It is important to remember that the claims principal factory is used only when a user signs in, which means that changes made to the claims in the role store do not take effect immediately. Request `http://localhost:5000/roles`, click the Edit button for the Administrator role, and create a Manager role claim, as shown in Figure 19-8.

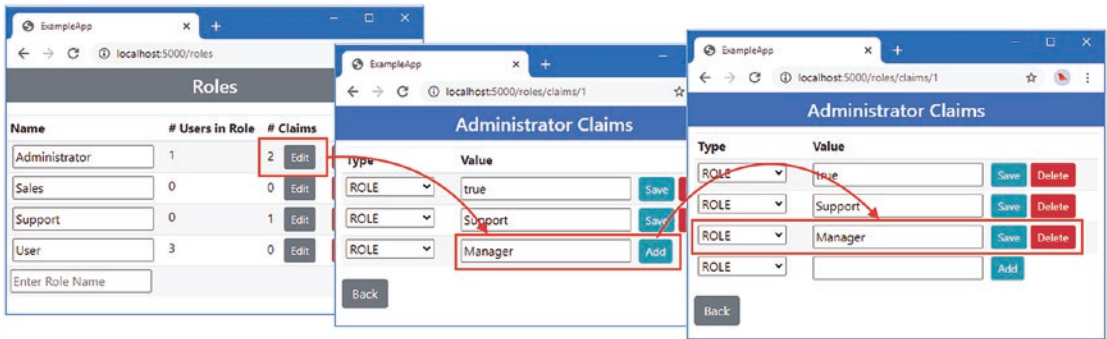


Figure 19-8. Adding a claim to a role

Navigate to `http://localhost:5000/liveclaims`, and you will see the list of claims for Alice does not include the new role. Sign out of the application, sign in again, and request `http://localhost:5000/liveclaims`. During the signing-in process, the claims principal factory creates a new `ClaimsPrincipal` object containing the new role claim, as shown in Figure 19-9.

role	User
role	Administrator
AccessUserData	true
role	Support
role	Manager
amr	pwd

Figure 19-9. The effect of signing in on claims

Summary

In this chapter, I created a role store. I described the features that role stores provides. I demonstrated how to validate roles, how to use the role store to ensure consistency in role assignments, and how to store claims. In the next chapter, I describe how lockouts and two-factor authentication are implemented.