CHAPTER 10

Using Roles and Claims

In this chapter, I describe the Identity API features that support roles and claims. Roles are commonly used to create fine-grained authorization policies that differentiate between different signed-in users. Claims are a general-purpose approach to describing any data that is known about a user and allow custom data to be added to the Identity user store. Table 10-1 puts these features in context.

Table 10-1. *Putting Roles and Claims in Context*

Question	Answer
What are they?	Roles differentiate groups of users to create authorization policies. Claims are used to represent any data known about a user.
Why are they useful?	Both features are used to provide the ASP.NET Core platform with data when a user signs in. Claims are not widely used, which is a shame because they make it easy to extend the data stored by Identity.
How are they used?	Users are assigned to roles, which are validated against a master list in an additional data repository, known as the *role store*. Claims are used to arbitrary data items, which are included in the data provided to the ASP.NET Core when a user signs in.
Are there any pitfalls or limitations?	It is important to prevent administrators from locking themselves out of the management tools by removing the role that grants them access.
Are there any alternatives?	Roles and claims are optional features. Roles are not required if you only need to identify signed-in users. Claims are not required if you do not need to store additional data.

Table 10-2 summarizes the chapter.

Table 10-2. *Chapter Summary*

Problem	Solution	Listing
Assign users to roles	Use the role manager to create a role. Use the user manager to assign the user to the role or to obtain the list of roles to which the user has already been assigned.	1–12
Restrict access using a role	Use the `Roles` argument of the `Authorize` attribute.	13
Assign claims to users	Use the user manager methods to create, delete, or replace claims, or to obtain a list of claims that have already been created.	14–17

(*continued*)

© Adam Freeman 2021
A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_10

Table 10-2. (*continued*)

Problem	Solution	Listing
Inspect the way the data in the user store will be provided to the ASP.NET Core platform	Use the sign-in manager to create a `ClaimsPrincipal` object.	18–19
Use claims in an ASP.NET Core application	Use the claims convenience methods provided by the `ClaimsPrincipal` class.	20

Preparing for This Chapter

This chapter uses the `IdentityApp` project from Chapter 9. Open a new PowerShell command prompt and run the commands shown in Listing 10-1 to reset the application and Identity databases.

■ **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 10-1. Resetting the Databases

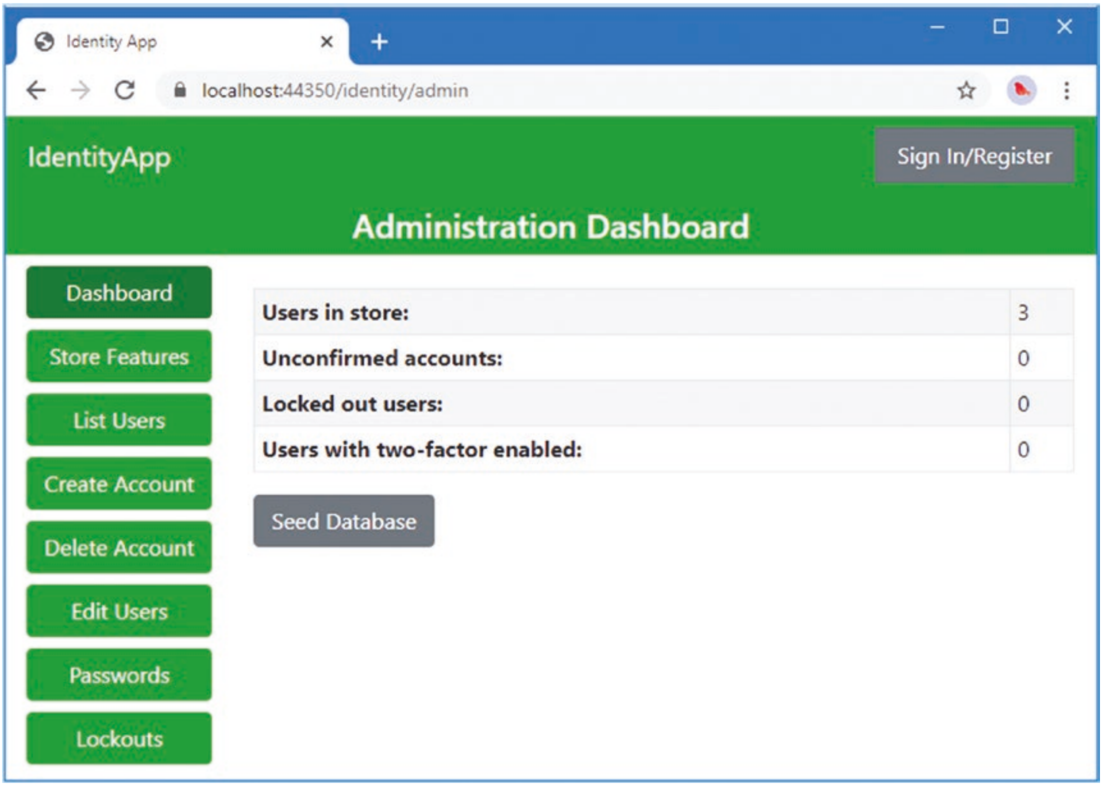
```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 10-2 in the `IdentityApp` folder to start the application.

Listing 10-2. Running the Example Application

```
dotnet run
```

Open a web browser and request `https://localhost:44350/Identity/Admin`, which will show the dashboard. Click the Seed Database button to add the test accounts to the user store, as shown in Figure 10-1.



Users in store:	3
Unconfirmed accounts:	0
Locked out users:	0
Users with two-factor enabled:	0

Figure 10-1. *Running the example application*

Using Roles

Roles are the most commonly used way to control access in applications. An authorization policy is defined that restricts access to one or more roles, such as the attribute I added to the most protected view and Razor Page in Chapter 3.

```
...
[Authorize(Roles = "Admin")]
public class AdminController : Controller {
...
```

Access is restricted to users that have been assigned the `Admin` role. The application isn't configured to support roles at present, and the Identity UI package doesn't support them at all, since they are not suitable for self-managed accounts, so there is no way to satisfy the authorization policy defined by the `Authorize` attribute. In the sections that follow, I set up support for roles and create the workflows for managing them.

Managing Roles

When you assign a user to a role, the user manager class asks the user store to add the role by providing it with a `string` to associate with the `IdentityUser` object but doesn't specify how this data is stored, which means that the implementation of roles can vary based on the user store you are working with.

In Part 2, I create a user store that keeps track of a user's roles using a collection so that the roles assigned to a user are stored independently of all other user roles. This approach is simple, but it is easy to introduce a typo so that Alice is assigned to the `admin` role, for example, but Bob is assigned to the `amdin` role. Mistakes like this can be hard to spot, and they prevent users' requests from being authorized correctly because roles names must match exactly.

One approach to avoiding this problem is to define a master list of roles and only allow users to be assigned to roles on that list. Identity has an advanced feature called the *role store*, which is used to associate extra data with roles. The role store doesn't change the way that a user's role assignments are stored, but it does provide a convenient master list to help ensure that roles are assigned consistently. (I describe the role store in detail, including its use as a master list of roles in Chapter 19.)

The master list is managed using the role manager class, `RoleManager<T>` where `T` is the role class used by the application. In Chapter 7, I enabled support for roles by selecting the default role class, `IdentityRole`, like this:

```
...
services.AddIdentity<IdentityUser, IdentityRole>(opts => {
    opts.Password.RequiredLength = 8;
    opts.Password.RequireDigit = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireNonAlphanumeric = false;
    opts.SignIn.RequireConfirmedAccount = true;
}).AddEntityFrameworkStores<IdentityDbContext>()
    .AddDefaultTokenProviders();
...
```

This means that the role manager class for the example application is `RoleManager<IdentityRole>`. The role manager class provides additional features, described in Chapter 19, that are not required by most projects, but members shown in Table 10-3 are the ones that are important for this chapter.

Table 10-3. *Important RoleManager<IdentityRole> Members*

Name	Description
`Roles`	This property returns an `IQueryable<IdentityRole>` that allows the roles in the role store to be enumerated or queried with LINQ.
`FindByNameAsync(name)`	This method locates the `IdentityRole` object with the specified name in the role store.
`CreateAsync(identRole)`	This method adds the specified `IdentityRole` object to the role store and returns an `IdentityResult` object that describes the outcome.
`DeleteAsync(identRole)`	This method removes the specified `IdentityRole` object from the role store and returns an `IdentityResult` object that describes the outcome.

These methods allow the master list of roles to be managed. Since we are using the `IdentityRole` class only as an entry on the master role list, only the property described in Table 10-4 is important for this chapter.

Table 10-4. *The IdentityRole Property*

Name	Description
`Name`	This property returns the name of the role.

The `RoleManager<IdentityRole>` class is used only for creating the master list of roles. The `UserManager<IdentityRole>` class defines the methods described in Table 10-5 to manage the assignment of users to roles from the master list.

Table 10-5. *The UserManager<IdentityRole> Methods for Managing Roles*

Name	Description
`GetRolesAsync(user)`	This method returns an `IList<string>` containing the names of all the roles to which the user has been assigned.
`IsInRoleAsync(user, name)`	This method returns `true` if the user has been assigned to the role with the specified name and `false` otherwise.
`AddToRoleAsync(user, name)`	This method assigns the user to the role with the specified name and returns an `IdentityResult` object that describes the outcome. There is also an `AddToRolesAsync` method that assigns the user to multiple roles at once.
`RemoveFromRoleAsync(user, name)`	This method removes the user from the role with the specified name and returns an `IdentityResult` object that describes the outcome. There is also a `RemoveFromRolesAsync` method that removes the user from multiple roles at once.

Add a Razor Page named `Roles.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 10-3.

Listing 10-3. The Contents of the Roles.cshtml File in the Pages/Identity/Admin Folder

```
@page "{id?}"
@model IdentityApp.Pages.Identity.Admin.RolesModel
@{
    ViewBag.Workflow = "Roles";
}

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

<table class="table table-sm table-striped table-bordered">
    <thead><tr><th colspan="2" class="text-center">Master Role List</th></tr></thead>
    <tbody>
        @foreach (IdentityRole role in Model.RoleManager.Roles) {
            int userCount =
                (await Model.UserManager.GetUsersInRoleAsync(role.Name)).Count;
            <tr>
                <td>@role.Name</td>
                <td>
                    @if (userCount == 0) {
                        <form method="post" asp-page-handler="deleteFromList">
                            <input type="hidden" name="role" value="@role.Name" />
                            <button type="submit" class="btn btn-sm btn-danger">
                                Delete
                            </button>
                        </form>
                    } else {
                        @: @userCount users in role
                    }
                </td>
            </tr>
        }
        <tr>
            <td>
                <form method="post" asp-page-handler="addToList" id="addToListForm">
                    <input class="form-control" name="role" />
                </form>
            </td>
            <td>
                <button type="submit" class="btn btn-sm btn-success"
                         form="addToListForm">
                     Add
                </button>
            </td>
        </tr>
    </tbody>
</table>

<table class="table table-sm table-striped table-bordered">
    <thead><tr><th colspan="2" class="text-center">User's Roles</th></tr></thead>
    <tbody>
        @if (Model.RoleManager.Roles.Count() == 0) {
            <tr>
                <td colspan="2" class="text-center py-2">
                    No roles have been defined
                </td>
            </tr>
        } else {
            @if(Model.CurrentRoles.Count() == 0) {
                <tr>
                    <td colspan="2" class="text-center py-2">
                        User has no roles
                    </td>
                </tr>
            } else {
                @foreach (string role in Model.CurrentRoles) {
                    <tr>
                        <td>@role</td>
                        <td>
                            <form method="post" asp-page-handler="delete">
                                <input type="hidden" asp-for="Id" />
                                <input type="hidden" name="role" value="@role" />
                                <button type="submit" class="btn btn-sm btn-danger">
                                    Delete
                                </button>
                            </form>
                        </td>
                    </tr>
                }
            }
            @if (Model.AvailableRoles.Count == 0) {
                <tr>
                    <td colspan="2" class="text-center py-2">
                        User is in all roles
                    </td>
                </tr>
            } else {
                <tr>
                    <td>
                        <select class="form-control" name="role" form="addForm">
                            <option selected disabled>Choose Role</option>
                            @foreach (string role in Model.AvailableRoles) {
                                <option>@role</option>
                            }
                        </select>
                    </td>
                    <td>
                        <form method="post" asp-page-handler="add" id="addForm">
                            <input type="hidden" asp-for="Id" />
                            <button type="submit" class="btn btn-sm btn-success">
                                Add
                            </button>
                        </form>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

The view part of this page is more complex than most other workflows because I am managing the master list and the role assignments for a user on a single page. The view presents a table containing a list of the available roles, along with a Delete button to remove the role from the store if no users have been assigned to it. There is also an `input` element and an Add button for adding new roles to the master list.

To assign the user to a role, a `select` element is populated with `option` elements for all the roles to which the user has not been assigned, along with a list of roles to which they have been assigned.

To define the page model class, add the code shown in Listing 10-4 to the `Roles.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 10-4. The Contents of the Roles.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class RolesModel : AdminPageModel {

        public RolesModel(UserManager<IdentityUser> userMgr,
                RoleManager<IdentityRole> roleMgr) {
            UserManager = userMgr;
            RoleManager = roleMgr;
        }

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

        public UserManager<IdentityUser> UserManager { get; set; }
        public RoleManager<IdentityRole> RoleManager { get; set; }

        public IList<string> CurrentRoles { get; set; } = new List<string>();
        public IList<string> AvailableRoles { get; set; } = new List<string>();

        private async Task SetProperties() {
            IdentityUser user = await UserManager.FindByIdAsync(Id);
            CurrentRoles = await UserManager.GetRolesAsync(user);
            AvailableRoles = RoleManager.Roles.Select(r => r.Name)
                .Where(r => !CurrentRoles.Contains(r)).ToList();
        }

        public async Task<IActionResult> OnGetAsync() {
            if (string.IsNullOrEmpty(Id)) {
                return RedirectToPage("Selectuser",
                    new { Label = "Edit Roles", Callback = "Roles" });
            }
            await SetProperties();
            return Page();
        }

        public async Task<IActionResult> OnPostAddToList(string role) {
            IdentityResult result =
                await RoleManager.CreateAsync(new IdentityRole(role));
            if (result.Process(ModelState)) {
                return RedirectToPage();
            }
            await SetProperties();
            return Page();
        }

        public async Task<IActionResult> OnPostDeleteFromList(string role) {
            IdentityRole idRole = await RoleManager.FindByNameAsync(role);
            IdentityResult result = await RoleManager.DeleteAsync(idRole);
            if (result.Process(ModelState)) {
                return RedirectToPage();
            }
            await SetProperties();
            return Page();
        }

        public async Task<IActionResult> OnPostAdd([Required] string role) {
            if (ModelState.IsValid) {
                IdentityResult result = IdentityResult.Success;
                if (result.Process(ModelState)) {
                    IdentityUser user = await UserManager.FindByIdAsync(Id);
                    if (!await UserManager.IsInRoleAsync(user, role)) {
                        result = await UserManager.AddToRoleAsync(user, role);
                    }
                    if (result.Process(ModelState)) {
                        return RedirectToPage();
                    }
                }
            }
            await SetProperties();
            return Page();
        }

        public async Task<IActionResult> OnPostDelete(string role) {
            IdentityUser user = await UserManager.FindByIdAsync(Id);
            if (await UserManager.IsInRoleAsync(user, role)) {
                await UserManager.RemoveFromRoleAsync(user, role);
            }
            return RedirectToPage();
        }
    }
}
```

The page model receives user manager and role manager objects through its constructor and uses them to populate the properties used to present the content in the view. The GET handler method locates the user object and using LINQ to work out which of the roles in the master list have already been assigned and which are still available. The other handlers add and remove roles to the role store and to add or remove a user from a role.

To integrate the new Razor Page, add the element shown in Listing 10-5 to the `_AdminWorkflows.cshtml` partial view.

Listing 10-5. Adding Navigation the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

```
@model (string workflow, string theme)

@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
}

<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-@Model.theme btn-block @getClass("Features")" asp-page="Features">
    Store Features
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Create")" asp-page="Create">
    Create Account
</a>
<a class="btn btn-success btn-block @getClass("Delete")" asp-page="Delete">
    Delete Account
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
<a class="btn btn-success btn-block @getClass("Passwords")" asp-page="Passwords"
        asp-route-id="">
    Passwords
</a>
<a class="btn btn-success btn-block @getClass("Lockouts")" asp-page="Lockouts" >
    Lockouts
</a>
<a class="btn btn-success btn-block @getClass("Roles")"
        asp-page="Roles" asp-route-id="">
    Edit Roles
</a>
```

Restart ASP.NET Core and request `https://localhost:44350/Identity/Admin`. Click the Edit Roles button and click the Roles button for the `alice@example.com` account. You will be presented with the content from the new Razor Page, which will be largely empty because there are no roles in the master list and the user has not been assigned any roles.

Enter Admin into the text field and click Add to add a new role to the master list. Now select Admin from the Choose Role select element and click the Add button to assign the selected user to the role. Notice that the Delete button for the role is removed once a user has been assigned to a role, as shown in Figure 10-2, because the role store won't allow a role to be deleted while it is in use. (This is a consequence of a foreign key relationship in the database used to store user and role data.)

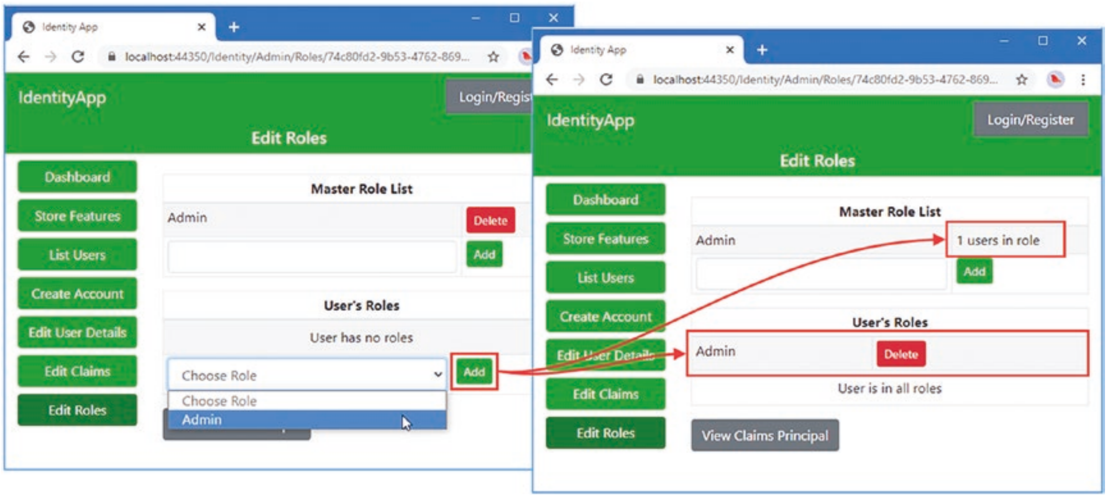
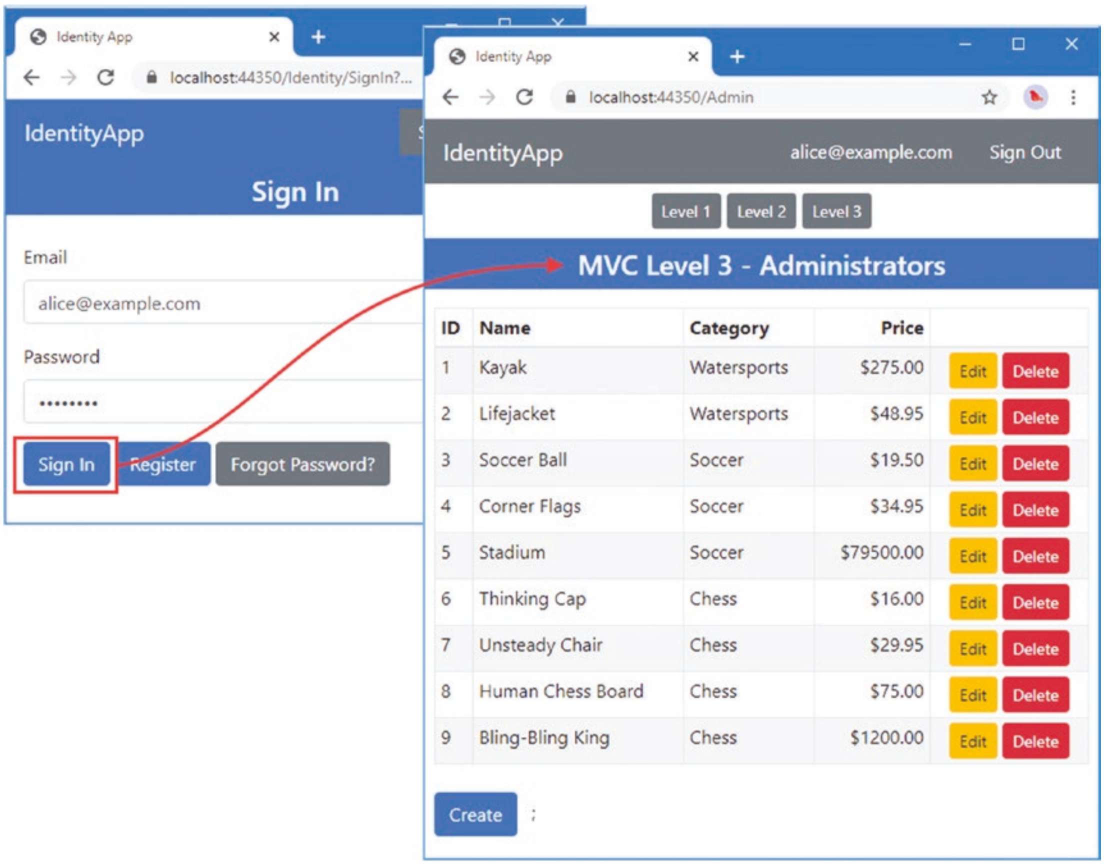


Figure 10-2. Assigning a user to a role

Testing the Role Membership

Until now, the application hasn't included support for assigning users to roles, which has meant that the most restricted level of content created in Chapter 3 has remained inaccessible. But now, the `alice@example.com` account has been assigned the `Admin` role, which means the user can access all of the content from Chapter 3.

Ensure you are signed out of the application and request `https://localhost:44350/admin`, which is one of the resources to which access is restricted by role. The request will generate a challenge response, and you will be redirected to the sign-in page. Sign in to the application using `alice@example.com` as the email address and `mysecret` as the password. Click the Sign In button, and you will be signed into the application and redirected to the restricted content, as shown in Figure 10-3.



ID	Name	Category	Price	
1	Kayak	Watersports	$275.00	Edit Delete
2	Lifejacket	Watersports	$48.95	Edit Delete
3	Soccer Ball	Soccer	$19.50	Edit Delete
4	Corner Flags	Soccer	$34.95	Edit Delete
5	Stadium	Soccer	$79500.00	Edit Delete
6	Thinking Cap	Chess	$16.00	Edit Delete
7	Unsteady Chair	Chess	$29.95	Edit Delete
8	Human Chess Board	Chess	$75.00	Edit Delete
9	Bling-Bling King	Chess	$1200.00	Edit Delete

Figure 10-3. *Accessing content protected by a role-based authorization policy*

■ **Caution** Changes to a user's roles do not take effect until the next time the user signs in.

Restricting Access to the Identity Administrator Dashboard

Now that the application has support for assigning roles, it is time to restrict access to the administrator dashboard. But, before doing that, some preparation is required to ensure that the administrator can sign in even after the database is reset and can't lock themselves out of the application once it is started.

Populating the User and Role Store During Startup

The first step is to ensure that the role that will grant access to the dashboard exists when the application starts and that there is a user account that has been assigned to that role. Add a class file named `DashboardSeed.cs` to the `IdentityApp` folder with the code shown in Listing 10-6.

Listing 10-6. The Contents of the DashboardSeed.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.Tasks;

namespace IdentityApp {
    public static class DashBoardSeed {

        public static void SeedUserStoreForDashboard(this IApplicationBuilder app) {
            SeedStore(app).GetAwaiter().GetResult();
        }

        private async static Task SeedStore(IApplicationBuilder app) {
            using (var scope = app.ApplicationServices.CreateScope()) {
                IConfiguration config =
                    scope.ServiceProvider.GetService<IConfiguration>();
                UserManager<IdentityUser> userManager =
                    scope.ServiceProvider.GetService<UserManager<IdentityUser>>();
                RoleManager<IdentityRole> roleManager =
                    scope.ServiceProvider.GetService<RoleManager<IdentityRole>>();

                string roleName = config["Dashboard:Role"] ?? "Dashboard";
                string userName = config["Dashboard:User"] ?? "admin@example.com";
                string password = config["Dashboard:Password"] ?? "mysecret";

                if (!await roleManager.RoleExistsAsync(roleName)) {
                    await roleManager.CreateAsync(new IdentityRole(roleName));
                }
                IdentityUser dashboardUser =
                    await userManager.FindByEmailAsync(userName);
                if (dashboardUser == null) {
                    dashboardUser = new IdentityUser {
                        UserName = userName,
                        Email = userName,
                        EmailConfirmed = true
                    };
                    await userManager.CreateAsync(dashboardUser);
                    dashboardUser = await userManager.FindByEmailAsync(userName);
                    await userManager.AddPasswordAsync(dashboardUser, password);
                }
                if (!await userManager.IsInRoleAsync(dashboardUser, roleName)) {
                    await userManager.AddToRoleAsync(dashboardUser, roleName);
                }
            }
        }
    }
}
```

The class defines an extension method that makes it easy to prepare the user store from the `Startup` class. A new dependency injection scope is created, and the role and user managers are used to ensure the required role and user exist. The name of the role, the name of the user, and the user's initial password are obtained from the ASP.NET Core configuration system so that the default values are easily overridden.

■ **Caution** There is a balance between making sure you don't lock yourself out of the application and creating a backdoor through which malicious access can be obtained. When you deploy a project, make sure you change the default passwords and consider requiring two-factor authentication for special accounts. The workflows for two-factor authentication are described in Chapter 11 and explained in detail in Chapter 22.

Listing 10-7 invokes the extension method to prepare the user store in the `Startup` class.

Listing 10-7. Preparing the User Store in the Startup.cs File in the IdentityApp Folder

```
...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });

    app.SeedUserStoreForDashboard();
}
...
```

Protecting the Dashboard Role

The next step is to make sure that there is at least one user assigned to the role that will grant access to the dashboard. The changes shown in Listing 10-8 obtain the name of the role from the configuration service and prevent the last user from being removed from the role.

Listing 10-8. Protecting the Role in the Roles.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;

namespace IdentityApp.Pages.Identity.Admin {

    public class RolesModel : AdminPageModel {

        public RolesModel(UserManager<IdentityUser> userMgr,
                RoleManager<IdentityRole> roleMgr,
                IConfiguration config) {
            UserManager = userMgr;
            RoleManager = roleMgr;
            DashboardRole = config["Dashboard:Role"] ?? "Dashboard";
        }

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

        public UserManager<IdentityUser> UserManager { get; set; }
        public RoleManager<IdentityRole> RoleManager { get; set; }

        public IList<string> CurrentRoles { get; set; } = new List<string>();
        public IList<string> AvailableRoles { get; set; } = new List<string>();

        public string DashboardRole { get; }

        // ...methods omitted for brevity...
    }
}
```

The changes use the configuration service to get the name of the dashboard role. To prevent the role from being removed, locate the delete button in the page's view and add the attribute shown in Listing 10-9.

Listing 10-9. Disabling a Button in the Roles.cshtml File in the Pages/Identity/Admin Folder

```
...
@foreach (string role in Model.CurrentRoles) {
    <tr>
        <td>@role</td>
        <td>
            <form method="post" asp-page-handler="delete">
                <input type="hidden" asp-for="Id" />
                <input type="hidden" name="role" value="@role" />
                <button type="submit" disabled="@(role== Model.DashboardRole)"
                        class="btn btn-sm btn-danger">
                    Delete
                </button>
            </form>
        </td>
    </tr>
}
...
```

The effect is to disable the delete button when the user is a member of the dashboard role.

■ **Note** Disabling the HTML element doesn't prevent someone from crafting an HTTP request that will delete the role. This would require appropriate authorization in the request, and the goal with these changes is just to prevent the administrator from accidentally locking themselves out of the application.

Protecting the Dashboard User

I need to make a corresponding change to prevent the dashboard account from being deleted. Listing 10-10 shows the changes to the Delete.cshtml file. For variety, I have used Razor expressions to disable the delete button without modifying the page model class.

Listing 10-10. Disabling a Button in the Delete.cshtml File in the Pages/Identity/Admin Folder

```
@page "{id?}"
@model IdentityApp.Pages.Identity.Admin.DeleteModel
@inject Microsoft.Extensions.Configuration.IConfiguration Configuration
@{
    ViewBag.Workflow = "Delete";
    string dashboardUser = Configuration["Dashboard:User"] ?? "admin@example.com";
}

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

<form method="post">
    <h3 class="bg-danger text-white text-center p-2">Caution</h3>
    <h5 class="text-center m-2">
        Delete @Model.IdentityUser.Email?
    </h5>
    <input type="hidden" name="id" value="@Model.IdentityUser.Id" />
    <div class="text-center p-2">
        <button type="submit" class="btn btn-danger"
            disabled="@(Model.IdentityUser.Email == dashboardUser)">
                Delete
        </button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

Updating the Test Account Seed Code

In Chapter 7, I added a handler method to the Dashboard page that deletes all the users in the store and replaces them with test accounts. In Listing 10-11, I have updated this method so that it does not remove users assigned to the role required to access the dashboard unless they are one of the test accounts.

Listing 10-11. Selecting Accounts in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using System.Linq;
using Microsoft.Extensions.Configuration;

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

        public DashboardModel(UserManager<IdentityUser> userMgr,
                IConfiguration configuration) {
            UserManager = userMgr;
            DashboardRole = configuration["Dashboard:Role"] ?? "Dashboard";
        }

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

        public string DashboardRole { get; set; }

        public int UsersCount { get; set; } = 0;
        public int UsersUnconfirmed { get; set; } = 0;
        public int UsersLockedout { get; set; } = 0;
        public int UsersTwoFactor { get; set; } = 0;

        private readonly string[] emails = {
            "alice@example.com", "bob@example.com", "charlie@example.com"
        };

        public void OnGet() {
            UsersCount = UserManager.Users.Count();
            UsersUnconfirmed = UserManager.Users
                .Where(u => !u.EmailConfirmed).Count();
            UsersLockedout = UserManager.Users
                .Where(u => u.LockoutEnabled
                    && u.LockoutEnd > System.DateTimeOffset.Now).Count();
        }

        public async Task<IActionResult> OnPostAsync() {
            foreach (IdentityUser existingUser in UserManager.Users.ToList()) {
               if (emails.Contains(existingUser.Email) ||
                     !await UserManager.IsInRoleAsync(existingUser, DashboardRole)) {
                    IdentityResult result
                         = await UserManager.DeleteAsync(existingUser);
                    result.Process(ModelState);
                }
            }
            foreach (string email in emails) {
                IdentityUser userObject = new IdentityUser {
                    UserName = email,
                    Email = email,
                    EmailConfirmed = true
                };
                IdentityResult result = await UserManager.CreateAsync(userObject);
                if (result.Process(ModelState)) {
                    result = await UserManager.AddPasswordAsync(userObject,
                        "mysecret");
                    result.Process(ModelState);
                }
                result.Process(ModelState);
            }
            if (ModelState.IsValid) {
                return RedirectToPage();
            }
            return Page();
        }
    }
}
```

Navigating Directly to the Administration Dashboard

The final preparatory change just makes it easier to reach the administrator dashboard by altering the link displayed in the page header, as shown in Listing 10-12.

Listing 10-12. Altering Navigation in the _LoginPartial.cshtml File in the Views/Shared Folder

```
@inject Microsoft.Extensions.Configuration.IConfiguration Configuration
@{
    string dashboardRole = Configuration["Dashboard:Role"] ?? "Dashboard";
}
<nav class="nav">
    @if (User.Identity.IsAuthenticated) {
        @if (User.IsInRole(dashboardRole)) {
            <a asp-page="/Identity/Admin/Dashboard"
                class="nav-link bg-secondary text-white">
                    @User.Identity.Name
            </a>
        } else {
            <a asp-page="/Identity/Index" class="nav-link bg-secondary text-white">
                    @User.Identity.Name
            </a>
        }
        <a asp-page="/Identity/SignOut" class="nav-link bg-secondary text-white">
            Sign Out
        </a>
    } else {
        <a asp-page="/Identity/SignIn" class="nav-link bg-secondary text-white">
            Sign In/Register
        </a>
    }
</nav>
```

Applying the Authorization Policy

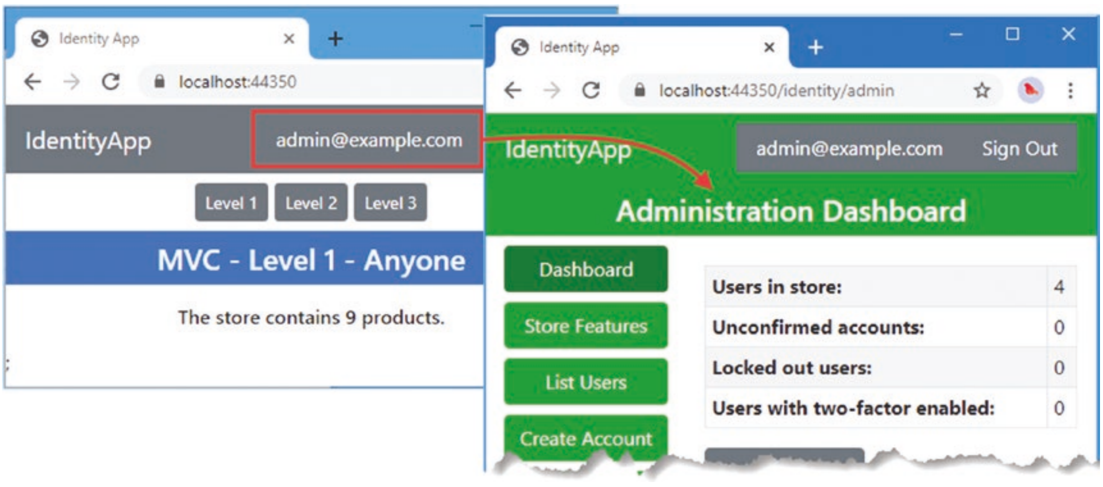
All of the preparations are in place, and all that remains is to apply the `Authorize` attribute to the common base class used by all the administrator dashboard pages, as shown in Listing 10-13.

■ **Tip** The roles feature of the `Authorize` attribute requires a literal string, which makes it difficult to read the role name from the configuration service. In Part 2, I explain how to create code-based authorization policies, which address this limitation.

Listing 10-13. Adding an Attribute in the AdminPageModel.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages.Identity.Admin {

    //[AllowAnonymous]
    [Authorize(Roles = "Dashboard")]
    public class AdminPageModel : UserPageModel {

        // no methods or properties required
    }
}
```

Restart ASP.NET Core and make sure you are signed out of the application. Sign in as `admin@example.com` using the password `mysecret`. Once you are signed in, click the email address at the top of the layout, and you will be presented with the administration dashboard, as shown in Figure 10-4.



MVC - Level 1 - Anyone

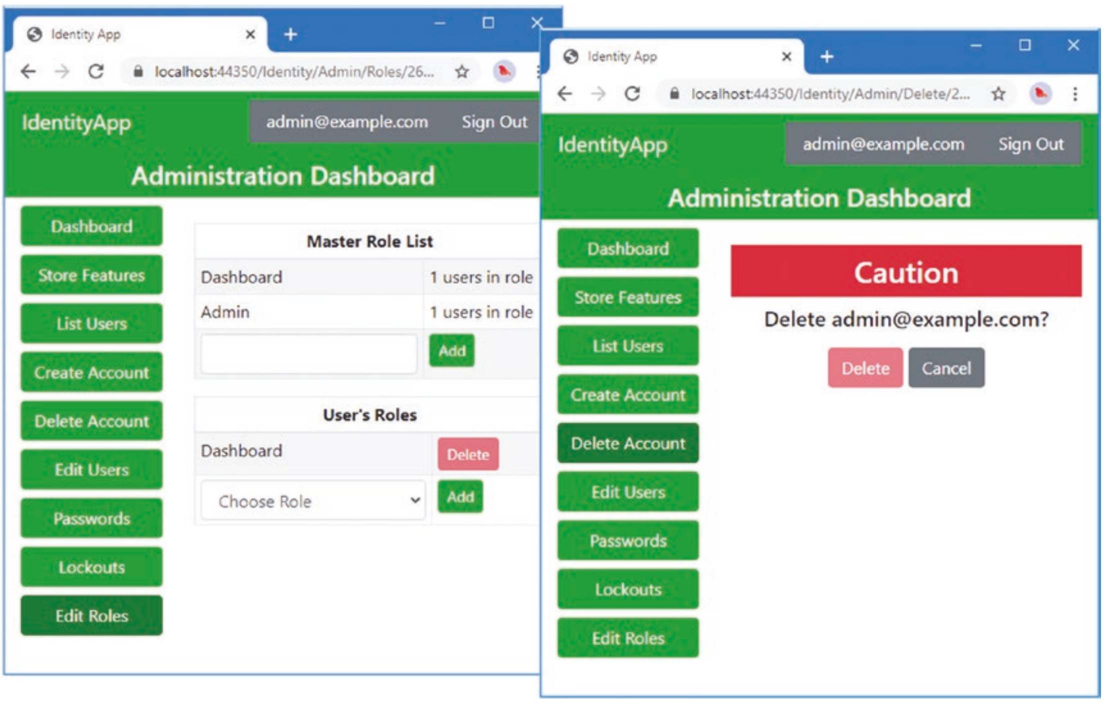
The store contains 9 products.

Administration Dashboard

Users in store:	4
Unconfirmed accounts:	0
Locked out users:	0
Users with two-factor enabled:	0

__Figure 10-4.__ Accessing the administrator dashboard

Click the Roles button and select the `admin@example.com` account; you will see that the Delete button for the membership of the `Dashboard` role is disabled, as shown in Figure 10-5. Click the Delete Account button and select the `admin@example.com` account; you will see the Delete button for the account is disabled, also shown in Figure 10-5.



Administration Dashboard

Master Role List	
Dashboard	1 users in role
Admin	1 users in role

User's Roles

Dashboard

Caution

Delete admin@example.com?

__Figure 10-5.__ Preventing accidental self-lockout

Managing Claims

A claim is a piece of data that is known about the user. This is a vague description because there are no limits placed on what a claim can describe or where it comes from. Claims are represented using the `Claim` class defined in the `System.Security.Claims` namespace. Table 10-6 describes the most important properties defined by the `Claim` class.

Table 10-6. *Important Claim Properties*

Name	Description
`Type`	This property returns the claim type.
`Value`	This property returns the claim value.
`Issuer`	This property returns the source of the claim.

An application can collect claims about a user from multiple sources and act on what the claims assert about the user and how much the source of the claim is trusted. For example, when determining if I should be granted access to confidential company data, an application may obtain claims from the HR system, the payroll system, and a social media platform. The HR system and the payroll system may claim that I am a junior programmer, but the social media platform may claim that I am the CEO. The application can give more weight to the claims from the systems that it trusts, discount the claims from sources it doesn't trust, and conclude that I am, in fact, a junior programmer with an ambitious public profile.

An application can also collect multiple identities for a single user. Different systems may use account names, for example, and so there may be different sets of claims for each identity.

In practice, many applications don't use claims directly at all, relying on roles to manage a user's access to restricted resources. Even when claims are used, applications tend not to have a nuanced understanding of a user's claims and identities because building and maintaining that understanding is complex. But, even so, understanding how claims are used is important because they underpin important Identity features, such as roles, and because they are the way that Identity expresses the data in the user store to the rest of the ASP.NET Core platform.

Making Claims Easier to Use

The type and value of a claim can be anything an application requires, which is great for flexibility but can be difficult to keep track of. To help keep claims consistent, Microsoft provides the `ClaimTypes` class, which defines commonly required claim types. These can be supplemented by custom types as needed.

Dealing with claims is made easier if you explicitly define the selection of claim types that you require. Add a class named `ApplicationClaimTypes.cs` to the `IdentityApp/Models` folder and add the code shown in Listing 10-14.

Listing 10-14. The Contents of the ApplicationClaimTypes.cs File in the Models Folder

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

namespace IdentityApp.Models {

    public static class ApplicationClaimTypes {
        public const string Country = ClaimTypes.Country;
        public const string SecurityClearance = "SecurityClearance";

        public static string GetDisplayName(this Claim claim)
            => GetDisplayName(claim.Type);

        public static string GetDisplayName(string claimType)
            => typeof(ClaimTypes).GetFields().Where(field =>
                    field.GetRawConstantValue().ToString() == claimType)
                        .Select(field => field.Name)
                        .FirstOrDefault() ?? claimType;

        public static IEnumerable<(string type, string display)> AppClaimTypes
            = new[] { Country, SecurityClearance }.Select(c =>
                (c, GetDisplayName(c)));
    }
}
```

I am going to add two types of claims to the user store, and I will refer to those types using the constants defined by the `ApplicationClaimTypes` class. The `Country` constant refers to the `ClaimTypes.Country` type, which is one of the types described by the `ClaimTypes` class provided by Microsoft. The `SecurityClearance` constant is a custom claim type, which I have defined with a standard .NET string.

The `ApplicationClaimTypes` class also defines an extension method for the `Claim` class, which will give me a value I can display to the user for a claim type. This is helpful because the claim types defined by the `ClaimTypes` class are expressed as URIs, which I don't want to display directly.

To manage the claims in the user store, add a Razor Page named `Claims.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 10-15.

Listing 10-15. The Contents of the Claims.cshtml File in the Pages/Identity/Admin Folder

```
@page "{id?}"
@model IdentityApp.Pages.Identity.Admin.ClaimsModel
@{
    ViewBag.Workflow = "Claims";
    int FormCounter = 0;
}

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

<table class="table table-sm table-bordered table-striped">
    <thead><tr><th>Type</th><th>Value</th><th>Issuer</th><th/></tr></thead>
    <tbody>
        @if (Model.Claims?.Count() > 0) {
            @foreach (Claim c in Model.Claims) {
                <tr>
                    <td>@c.GetDisplayName()</td>
                    <td>
                        <form method="post" id="@(++FormCounter)">
                            <input type="hidden" asp-for="Id" />
                            <input type="hidden" name="type" value="@c.Type" />
                            <input type="hidden" name="oldValue" value="@c.Value" />
                            <input class="form-control" name="value"
```

```
                                value="@c.Value" />
                        </form>
                    </td>
                    <td>@c.Issuer</td>
                    <td>
                        <button class="btn btn-sm btn-warning" form="@(FormCounter)"
                                asp-route-task="change">Change</button>
                        <button class="btn btn-sm btn-danger" form="@(FormCounter)"
                                asp-route-task="delete">Delete</button>
                    </td>
                </tr>
            }
        } else {
            <tr><th colspan="4" class="text-center py-3">User has no claims</th></tr>
        }
        </tbody>
        <tfoot>
            <tr><th colspan="4" class="text-center pt-3">Add New Claim</th></tr>
            <tr>
                <td>
                    <form method="post" id="addClaim" asp-route-task="add">
                        <select class="form-control" name="type">
                            @foreach (var claimType in
                                    ApplicationClaimTypes.AppClaimTypes) {
                                <option value="@claimType.type">
                                     @claimType.display
                                </option>
                            }
                        </select>
                    </form>
                </td>
                <td colspan="2">
                    <input class="form-control" form="addClaim" name="value" />
                </td>
                <td>
                    <button type="submit" form="addClaim"
                        class="btn btn-sm btn-success">Add</button>
                </td>
            </tr>
        </tfoot>
</table>

<a asp-page="ViewClaimsPrincipal" class="btn btn-secondary"
        asp-route-id="@Model.Id" asp-route-callback="Claims">
    View Claims Principal
</a>
```

The view section of the page presents a table that contains the user's claims. Each row contains the claim type, value, and issuer, along with buttons that will save changes to the claim's value or remove the claim from the store. There is also a form that allows a new claim to be created and a View Claims Principal button that targets a page that I create in the next section. To define the page model class, add the code shown in Listing 10-16 to the `Claims.cshtml.cs`. (You will have to create this file if you are using Visual Studio Code.)

Listing 10-16. The Contents of the Claims.cshtml.cs File in the Pages/Identity/Admin Folder

```
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;

namespace IdentityApp.Pages.Identity.Admin {

    public class ClaimsModel : AdminPageModel {

        public ClaimsModel(UserManager<IdentityUser> mgr)
            => UserManager = mgr;

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

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

        public IEnumerable<Claim> Claims { get; set; }

        public async Task<IActionResult> OnGetAsync() {
            if (string.IsNullOrEmpty(Id)) {
                return RedirectToPage("Selectuser",
                    new { Label = "Manage Claims", Callback = "Claims" });
            }
            IdentityUser user = await UserManager.FindByIdAsync(Id);
            Claims = await UserManager.GetClaimsAsync(user);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync([Required] string task,
                [Required] string type, [Required] string value, string oldValue) {
            IdentityUser user = await UserManager.FindByIdAsync(Id);
            Claims = await UserManager.GetClaimsAsync(user);
            if (ModelState.IsValid) {
                Claim claim = new Claim(type, value);
                IdentityResult result = IdentityResult.Success;
                switch (task) {
                    case "add":
                        result = await UserManager.AddClaimAsync(user, claim);
                        break;
                    case "change":
                        result = await UserManager.ReplaceClaimAsync(user,
                            new Claim(type, oldValue), claim);
                        break;
                    case "delete":
                        result = await UserManager.RemoveClaimAsync(user, claim);
                        break;
                };
                if (result.Process(ModelState)) {
                    return RedirectToPage();
                }
            }
            return Page();
        }
    }
}
```

The GetClaimsAsync method is used to obtain the existing claims, and the AddClaimAsync, ReplaceClaimAsync, and RemoveClaimAsync methods are used to make changes. These methods automatically update the user store. For quick reference, Table 10-7 describes the methods used in the claim workflow. (See Chapter 17 for a detailed explanation of the user store support for claims.)

Table 10-7. *The UserManager<IdentityUser> Methods for Working with Claims*

Name	Description
`GetClaimsAsync(user)`	This method returns an `IList<Claim>` containing the claims in the user store for the specified user.
`AddClaimAsync(user, claim)`	This method adds a claim to the user store for the specified user.
`ReplaceClaimAsync(user, old, new)`	This method replaces one claim with another for the specified user.
`RemoveClaimAsync(user, claim)`	This method removes a claim from the user store for the specified user.

Add the element shown in Listing 10-17 to integrate the new page into the navigation partial view.

Listing 10-17. Adding an Element in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

```
@model (string workflow, string theme)

@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
}

<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-@Model.theme btn-block @getClass("Features")" asp-page="Features">
    Store Features
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Create")" asp-page="Create">
    Create Account
</a>
```

```
<a class="btn btn-success btn-block @getClass("Delete")" asp-page="Delete">
    Delete Account
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
<a class="btn btn-success btn-block @getClass("Passwords")" asp-page="Passwords"
        asp-route-id="">
    Passwords
</a>
<a class="btn btn-success btn-block @getClass("Lockouts")" asp-page="Lockouts" >
    Lockouts
</a>
<a class="btn btn-success btn-block @getClass("Roles")"
        asp-page="Roles" asp-route-id="">
    Edit Roles
</a>
<a class="btn btn-success btn-block @getClass("Claims")"
        asp-page="Claims" asp-route-id="">
    Claims
</a>
```

Restart ASP.NET Core, make sure you are signed in as `admin@example.com` using the password `mysecret`, and request `https://localhost:44350/Identity/Admin`.

Click the Claims button and select the `alice@example.com` account. The output from the `Claims` page shows there are no claims stored for this account.

Select Country from the `select` element in the Type column and enter USA into the Value field. Click the Add button to submit the form, and a new claim will be added to the store, as shown in Figure 10-6. Notice that when you create a claim, the `Issuer` property is set to `LOCAL_AUTHORITY`, which is the default value and denotes a claim originating within the application.

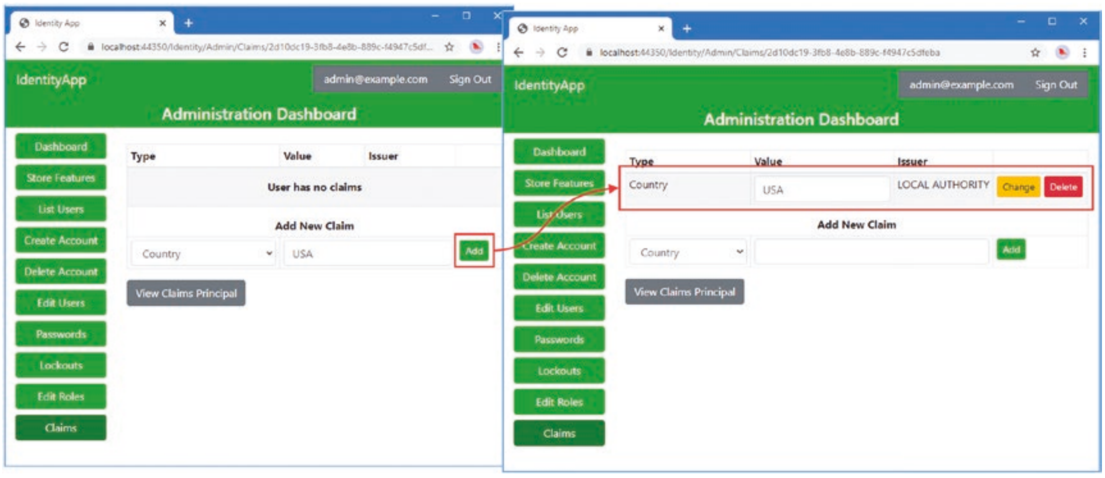
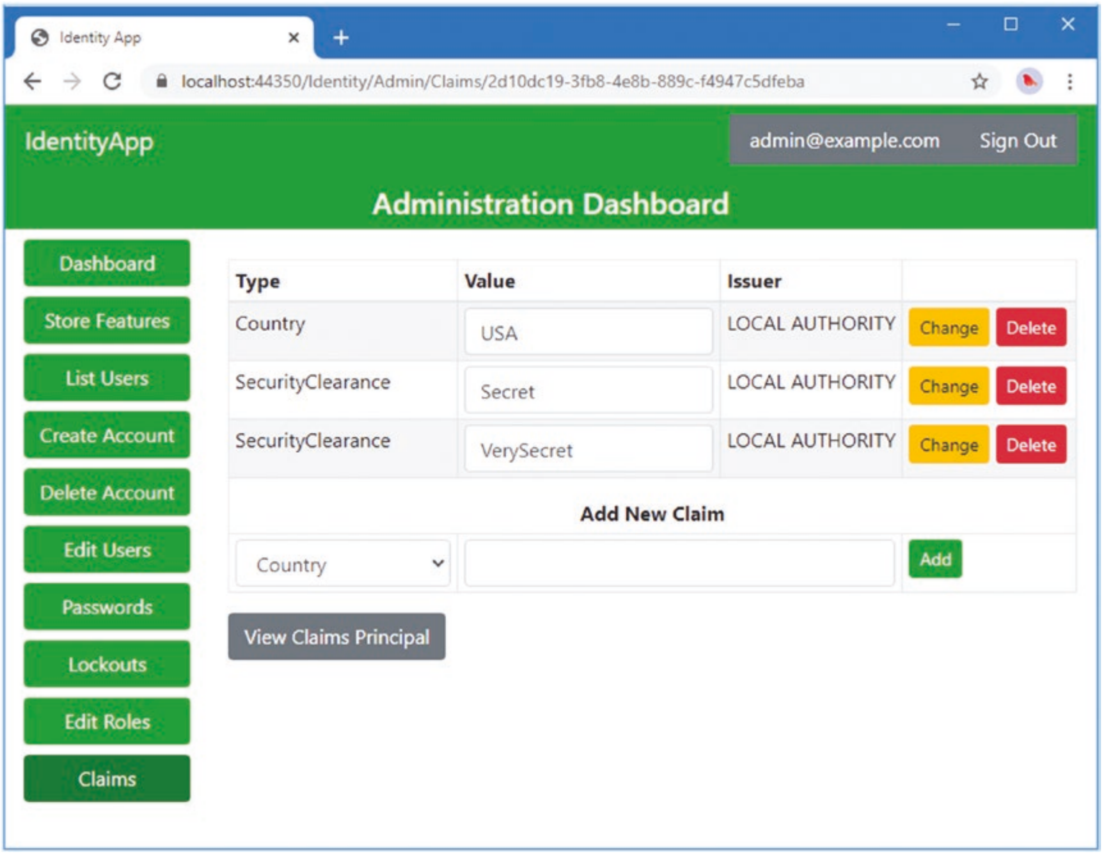


Figure 10-6. *Creating a claim*

The user store can contain multiple claims with the same type for the same user. Select the `SecurityClearance` claim type from the select element, enter Secret into the text field, and click Add. Repeat the process, entering VerySecret into the text field. When you click Add, a second `SecurityClearance` claim is added to the user store, as shown in Figure 10-7.



Type	Value	Issuer	
Country	USA	LOCAL AUTHORITY	Change Delete
SecurityClearance	Secret	LOCAL AUTHORITY	Change Delete
SecurityClearance	VerySecret	LOCAL AUTHORITY	Change Delete

Figure 10-7. *Creating multiple claims with the same type*

Providing ASP.NET Core with Claims Data

Adding claims to the user store is helpful only if they can be used in the rest of the application. The key to this is the `ClaimsPrincipal` object that is created when a user signs into the application to represent the current user and that can be accessed in Razor Pages and MVC controllers.

The `ClaimsPrincipal` object is usually created by the `SignInManager<IdentityUser>` class during the sign-in workflow, but you can create one directly using the `CreateUserPrincipalAsync` method. In Part 2, I explain the process by which `ClaimsPrincipal` objects are created and show you how to customize it.

Add a Razor Page named `ViewClaimsPrincipal.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 10-18.

Listing 10-18. The Contents of the ViewClaimsPrincipal.cshtml File in the Pages/Identity/Admin Folder

```
@page "{id?}/{callback?}"
@model IdentityApp.Pages.Identity.Admin.ViewClaimsPrincipalModel
@{
    ViewBag.Workflow = "ClaimsPrincipal";
    ViewBag.WorkflowLabel = "View ClaimsPrincipal";
    int counter = 0;
}

@foreach (ClaimsIdentity ident in Model.Principal.Identities) {
    <table class="table table-sm table-striped table-bordered pt-3">
        <thead>
            <tr><th colspan="3" class="text-center">Identity #@(++counter)</th></tr>
        </thead>
        <tbody>
            <tr><th>Type</th><th>Value</th><th>Issuer</th></tr>
            @foreach (Claim c in ident.Claims) {
                <tr>
                   <td>@c.GetDisplayName()</td>
                   <td class="text-truncate" style="max-width:250px">@c.Value</td>
                   <td>@c.Issuer</td>
                </tr>
            }
        </tbody>
    </table>
}

@if (!string.IsNullOrEmpty(Model.Callback)) {
    <a asp-page="@Model.Callback" class="btn btn-secondary" asp-route-id="@Model.Id">
        Back
    </a>
}
```

The view part of the page obtains a `ClaimsPrincipal` object from the page model and enumerates each user identity it represents. For each user identity, it displays all of the claim types, values, and issuers. (I am afraid there is no avoiding the multiple uses of the term *identity* in this section, reflecting a group of related claims for a user as well as the framework that is the subject of this book.)

The `ClaimsPrincipal` class defines the `Identities` property, which returns a sequence of `ClaimsIdentity` objects, representing the user's identities. The `ClaimsIdentity` class defines the `Claims` property, which returns a sequence of `Claim` objects. I describe additional features in later examples and in Part 2, but this basic set of properties is enough to display the set of claims that will be generated by ASP.NET Core Identity when the user signs into the application.

To define the page model class, add the code shown in Listing 10-19 to the `ViewClaimsPrincipal.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

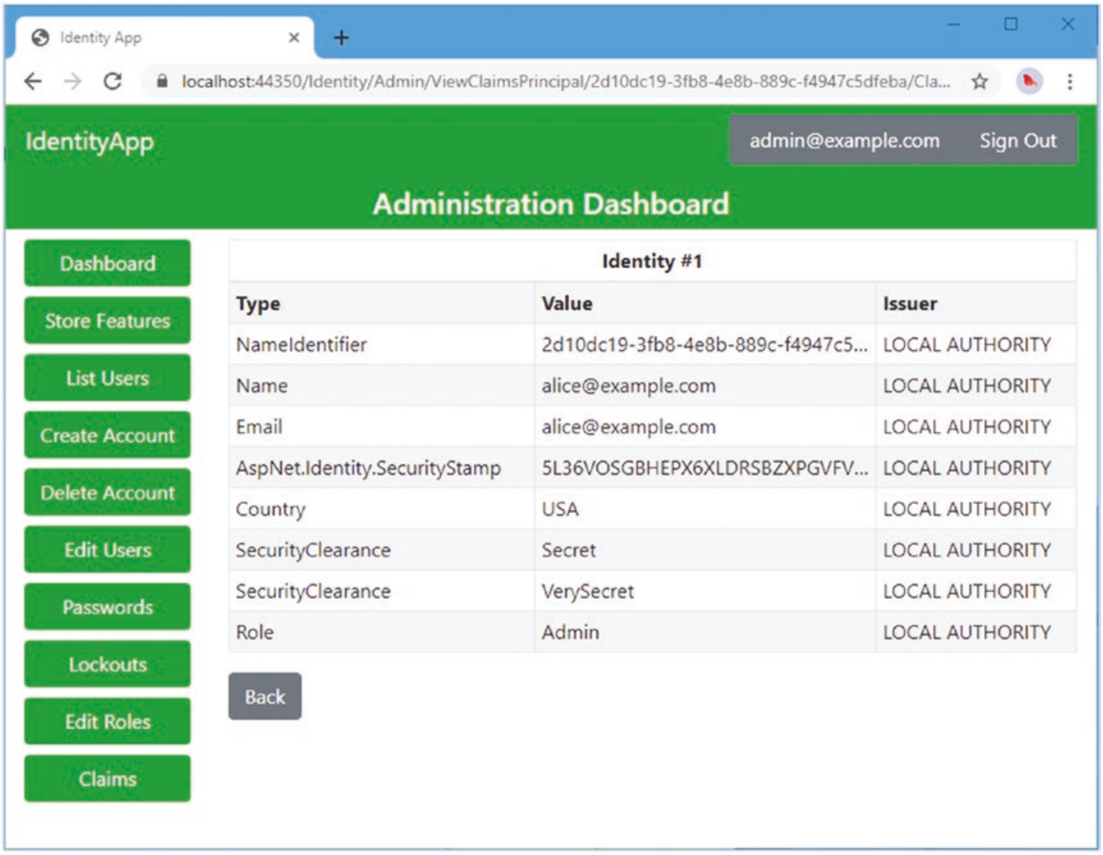
Listing 10-19. The Contents of the ViewClaimsPrincipal.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using System.Threading.Tasks;
```

```
namespace IdentityApp.Pages.Identity.Admin {

    public class ViewClaimsPrincipalModel : AdminPageModel {

        public ViewClaimsPrincipalModel(UserManager<IdentityUser> usrMgr,
                SignInManager<IdentityUser> signMgr) {
            UserManager = usrMgr;
            SignInManager = signMgr;
        }

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

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

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

        public ClaimsPrincipal Principal { get; set; }

        public async Task<IActionResult> OnGetAsync() {
            if (string.IsNullOrEmpty(Id)) {
                return RedirectToPage("Selectuser",
                    new {
                        Label = "View ClaimsPrincipal",
                        Callback = "ClaimsPrincipal"
                    });
            }
            IdentityUser user = await UserManager.FindByIdAsync(Id);
            Principal = await SignInManager.CreateUserPrincipalAsync(user);
            return Page();
        }
    }
}
```

The page model class obtains an `IdentityUser` class from the user store and uses the `Create` method defined by the `CreateUserPrincipalAsync` method defined by the sign-in manager class to produce the `ClaimsPrincipal` object required by the view.

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, and click the Edit Claims button. Click the Claims button to select the `alice@example.com` account and then click the View Claims Principal button, which will show how the data in the user store will be presented to the rest of the ASP.NET Core platform, as shown in Figure 10-8.



Identity #1

Type	Value	Issuer
NameIdentifier	2d10dc19-3fb8-4e8b-889c-f4947c5...	LOCAL AUTHORITY
Name	alice@example.com	LOCAL AUTHORITY
Email	alice@example.com	LOCAL AUTHORITY
AspNet.Identity.SecurityStamp	5L36VOSGBHEPX6XLDRSBZXPGVFV...	LOCAL AUTHORITY
Country	USA	LOCAL AUTHORITY
SecurityClearance	Secret	LOCAL AUTHORITY
SecurityClearance	VerySecret	LOCAL AUTHORITY
Role	Admin	LOCAL AUTHORITY

Figure 10-8. *Viewing the ClaimsPrincipal data*

There are three groups of claims for this user. The first group is created using selected properties defined by the `IdentityUser` class, including the `Id`, `UserName`, and `Email` properties, which are expressed with the `NameIdentifier`, `Name`, and `Email` claim types from the `ClaimTypes` class. This group also contains a claim for the user's `SecurityStamp`, but there is no `ClaimTypes` value for this type of claim and so the type `AspNet.Identity.SecurityStamp` is used.

The second group contains the `Country` and `SecurityClearance` claims added to the store in the previous section. The user's stored claims are added to the `ClaimsPrincipal` object without any modification.

The final group of claims describes the roles to which the user has been assigned. Each role is described with a claim whose type is set using the `ClaimTypes.Role` value and whose value is the name of the role.

These groups of claims describe all the data Identity has stored about a user in a way that the rest of the ASP.NET Core platform can use. However, since the data in the user store is presented to ASP.NET Core during the sign-in process, changes that you make to the user store—such as new role assignments and claims—won't take effect until the next time the user signs in.

Using Claims Data

Claims can be used in the rest of the application. In Part 2, for example, I explain how to use claims to create an authorization policy, although this is something that is usually done using roles, as demonstrated earlier in this chapter.

Add a Razor Page named Clearance.cshtml to the IdentityApp/Pages folder with the content shown in Listing 10-20.

Listing 10-20. The Contents of the Clearance.cshtml File in the Pages Folder

```
@page

@{
    Func<string, bool> HasClearance = (string level)
        => User.HasClaim(ApplicationClaimTypes.SecurityClearance, level);
}

<table class="table table-sm table-striped table-bordered">
    <thead><tr><th>Clearance Level</th><th>Granted</th></tr></thead>
    <tbody>
        <tr><td>Secret</td><td>@HasClearance("Secret")</td></tr>
        <tr><td>Very Secret</td><td>@HasClearance("VerySecret")</td></tr>
        <tr><td>Super Secret</td><td>@HasClearance("SuperSecret")</td></tr>
    </tbody>
</table>
```

Razor Pages and the MVC Framework provide access to the ClaimsPrincipal object for the current request with a User property that is available in views and the base classes used for page model and controller classes.

In Listing 10-20, I defined a function named HasClearance that reads the value of the User property to get the ClaimsPrincipal object and uses the HasClaim method to check to see if the user has a claim with a specific type and value. The HasClaim method checks all of the identities associated with the ClaimsPrincipal and is one of a set of convenience members for working with claims, as described in Table 10-8.

Table 10-8. *The ClaimsPrincipal Convenience Members for Claims*

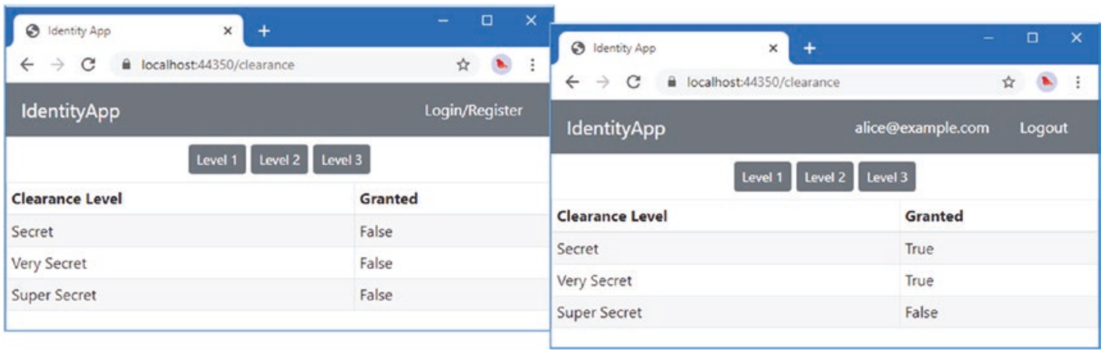
Name	Description
Claims	This property returns an IEnumerable<Claim> containing the claims from all the ClaimIdentity objects associated with the ClaimsPrincipal.

FindAll(type)
FindFirst(type)
	This method returns all of the claims, or the first claim, with the specified type from all the ClaimIdentity objects associated with the ClaimsPrincipal.

FindAll(filter)
FindFirst(filter)
	This method returns the claims, or the first claim, that matches the specified filter predicate from all the ClaimIdentity objects associated with the ClaimsPrincipal.

HasClaim(type, value)
HasClaim(filter)
	This method returns true if any of the ClaimIdentity objects associated with the ClaimsPrincipal has a claim with the specified type and value or that matches the specified predicate.

The members in Table 10-8 are convenient because they operate across all of the claims associated with the `ClaimsPrincipal`, regardless of how many `ClaimsIdentity` objects have been created. In Listing 10-20, I used the `HasClaim` method to determine if the user has `SecurityClearance` claims with specific values. Restart ASP.NET Core, make sure you are signed out of the application, and request `https://localhost:44350/clearance`. ASP.NET Core always associated a `ClaimsPrincipal` object with requests, even when there is no user signed into the application. When you signed out of the application, there will be no claims, and you will see the response shown on the left of Figure 10-9.



Clearance Level	Granted
Secret	False
Very Secret	False
Super Secret	False

Clearance Level	Granted
Secret	True
Very Secret	True
Super Secret	False

Figure 10-9. *Using claims*

Sign in to the application as `alice@example.com` and request `https://localhost:44350/clearance` again. This time the response will reflect the claims added to the store earlier, as shown on the right of Figure 10-9. (If you don't see the expected claims, then you may need to sign out of the application and sign in again. As I explained earlier, changes to the user store won't take effect until the next time the user signs in.)

Summary

In this chapter, I described the Identity support for roles and claims. Roles are the most common way of creating fine-grained authorization policies, but care has to be taken not to lock everyone out of the application or to create unexpected results with a typo. Claims are a general-purpose mechanism for describing data known about a user, and I explained how they are handled by Identity and how the data in the user store is expressed as a series of claims to the rest of the ASP.NET Core platform. In the next chapter, I describe the Identity support for two-factor authentication and external authentication services.