CHAPTER 12

SportsStore: Security

In the previous chapter, I added support for administering the SportsStore application, and it probably did not escape your attention that anyone could modify the product catalog if I deploy the application as it is. All they would need to know is that the administration features are available using the `/Admin/Index` and `/Order/List` URLs. In this chapter, I am going to show you how to prevent random people from using the administration functions by password-protecting them.

Securing the Administration Features

Authentication and authorization are provided by the ASP.NET Core Identity system, which integrates neatly into both the ASP.NET Core platform and MVC applications. In the sections that follow, I will create a basic security setup that allows one user, called `Admin`, to authenticate and access the administration features in the application. ASP.NET Core Identity provides many more features for authenticating users and authorizing access to application features and data, and you can find a more detailed information in Chapters 28–30, where I show you how to create and manage user accounts, how to use roles and policies, and how to support authentication from third parties, such as Microsoft, Google, Facebook, and Twitter. In this chapter, however, my goal is just to get enough functionality in place to prevent customers from being able to access the sensitive parts of the SportsStore application and, in doing so, give you a flavor of how authentication and authorization fit into an MVC application.

Adding the Identity Package to the Project

The first step is to add ASP.NET Identity to the SportsStore project, which requires some new NuGet packages. Listing 12-1 shows the additions to the `SportsStore.csproj` file in the SportsStore project.

Listing 12-1. Adding ASP.NET Core Identity in the SportsStore.csproj File of the SportsStore Project

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="Views\Admin\" />
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
        Version="2.0.0" />
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
        Version="1.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
        Version="1.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
        Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
        Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json"
        Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Session" Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Caching.Memory"
        Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Http.Extensions"
        Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
        Version="1.1.1" />

    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
       Version="1.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
       Version=" 1.0.0" />
  </ItemGroup>

</Project>
```

When the changes to the file are saved, Visual Studio will use NuGet to download and install the Identity package.

Creating the Identity Database

The ASP.NET Identity system is endlessly configurable and extensible and supports lots of options for how its user data is stored. I am going to use the most common, which is to store the data using Microsoft SQL Server accessed using Entity Framework Core.

Creating the Context Class

I need to create a database context file that will acts as the bridge between the database and the Identity model objects it provides access to. I added a class file called `AppIdentityDbContext.cs` to the `Models` folder and used it to define the class shown in Listing 12-2.

Listing 12-2. The Contents of the AppIdentityDbContext.cs File in the Models Folder

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {

    public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {

        public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options)
            : base(options) { }
    }
}
```

This class is derived from `IdentityDbContext`, which provides Identity-specific features for Entity Framework Core. For the type parameter, I used the `IdentityUser` class, which is the built-in class used to represent users. In Chapter 28, I demonstrate how to use a custom class that you can extend to add extra information about the users of your application.

Defining the Connection String

The next step is to define the connection string that will be for the database. In Listing 12-3, you can see the additions I made to the `appsettings.json` file of the SportsStore project, which follows the same format as the connection string that I defined for the product database in Chapter 8.

Listing 12-3. Defining a Connection String in the appsettings.json File

```
{
```

```
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
"Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;Trusted_Connection=True;Multiple
ActiveResultSets=true"
    },
    "SportStoreIdentity": {
      "ConnectionString":
"Server=(localdb)\\MSSQLLocalDB;Database=Identity;Trusted_Connection=True;MultipleAct
iveResultSets=true"
    }
  }
}
```

Remember that the connection string has to be defined in a single unbroken line in the `appsettings.json` file and is shown across multiple lines in the listing only because of the fixed width of a book page. The addition in the listing defines a connection string called `SportsStoreIdentity` that specifies a LocalDB database called `Identity`.

Configuring the Application

Like other ASP.NET Core features, Identity is configured in the `Start` class. Listing 12-4 shows the additions I made to set up Identity in the SportsStore project, using the context class and connection string defined previously.

Listing 12-4. Configuring Identity in the Startup.cs File

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {
        IConfigurationRoot Configuration;

        public Startup(IHostingEnvironment env) {
            Configuration = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json").Build();
```

```
        }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));

            services.AddDbContext<AppIdentityDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdentity:ConnectionString"]));

            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<AppIdentityDbContext>();

            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
            services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
            services.AddTransient<IOrderRepository, EFOrderRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }

        public void Configure(IApplicationBuilder app,
                IHostingEnvironment env, ILoggerFactory loggerFactory) {

            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseIdentity();
            app.UseMvc(routes => {

                // ...routes omitted for brevity...

            });
            SeedData.EnsurePopulated(app);
            IdentitySeedData.EnsurePopulated(app);
        }
    }
}
```

In the `ConfigureServices` method, I extended the Entity Framework Core configuration to register the context class and used the `AddIdentity` method to set up the Identity services using the built-in classes to represent users and roles. In the `Configure` method, I called the `UseIdentity` method to set up the components that will intercept requests and responses to implement the security policy. I also added a call to an `IdentitySeedData.EnsurePopulated` method, which I will create in the next section to add the user data to the database.

Defining the Seed Data

I am going to explicitly create the `Admin` user by seeding the database when the application starts. I added a class file called `IdentitySeedData.cs` to the `Models` folder and defined the static class shown in Listing 12-5.

Listing 12-5. The Contents of the IdentitySeedData.cs File in the Models Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123$";

        public static async void EnsurePopulated(IApplicationBuilder app) {

            UserManager<IdentityUser> userManager = app.ApplicationServices
                .GetRequiredService<UserManager<IdentityUser>>();

            IdentityUser user = await userManager.FindByIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                await userManager.CreateAsync(user, adminPassword);
            }
        }
    }
}
```

This code uses the `UserManager<T>` class, which is provided as a service by ASP.NET Core Identity for managing users, as described in Chapter 28. The database is searched for the `Admin` user account, which is created—with a password of `Secret123$`—if it is not present. Do not change the hard-coded password in this example because Identity has a validation policy that requires passwords to contain a number and range of characters. See Chapter 28 for details of how to change the validation settings.

Caution Hard-coding the details of an administration account is often required so that you can log into an application once it has been deployed and start administering it. When you do this,

you must remember to change the password for the account you have created. See Chapter 28 for details of how to change passwords using Identity.

Creating and Applying the Database Migration

All of the components are in place, and it is time to use the Entity Framework Core migrations feature to define the schema and apply it to the database. Open the Package Manager Console and run the following command to create the migration:

```
Add-Migration Initial -Context AppIdentityDbContext
```

The important difference from previous database commands is that I have used the `-Context` option to specify the name of the context class associated with the database that I want to work with, which is `AppIdentityDbContext`. When you have multiple databases in the application, it is important to ensure that you are working with the right one.

Once Entity Framework Core has generated the initial migration, run the following command to create the database and run the migration commands:

```
Update-Database -Context AppIdentityDbContext
```

The result is a new LocalDB database called `Identity` that you can inspect using the Visual Studio SQL Server Object Explorer.

Applying a Basic Authorization Policy

Now that I have installed and configured ASP.NET Core Identity, I can apply an authorization policy to the parts of the application that I want to protect. I am going to use the most basic authorization policy possible, which is to allow access to any authenticated user. Although this can be a useful policy in real applications as well, there are also options for creating finer-grained authorization controls (as described in Chapters 28–30), but since the SportsStore application has only one user, distinguishing between anonymous and authenticated requests is sufficient.

The `Authorize` attribute is used to restrict access to action methods, and in Listing 12-6, you can see that I have used the attribute to protect access to the administration actions in the `Order` controller.

Listing 12-6. Restricting Access in the OrderController.cs File

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;

        public OrderController(IOrderRepository repoService, Cart cartService) {
            repository = repoService;
            cart = cartService;
        }

        [Authorize]
        public ViewResult List() =>
            View(repository.Orders.Where(o => !o.Shipped));

        [HttpPost]
        [Authorize]
        public IActionResult MarkShipped(int orderID) {
            Order order = repository.Orders
                .FirstOrDefault(o => o.OrderID == orderID);
            if (order != null) {
                order.Shipped = true;
                repository.SaveOrder(order);
            }
            return RedirectToAction(nameof(List));
        }

        public ViewResult Checkout() => View(new Order());

        [HttpPost]
        public IActionResult Checkout(Order order) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
            if (ModelState.IsValid) {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Completed));
            } else {
                return View(order);
            }
        }
```

```
        public ViewResult Completed() {
            cart.Clear();
            return View();
        }
    }
}
```

I don't want to stop unauthenticated users from accessing the other action methods in the `Order` controller, so I have applied the `Authorize` attribute only to the `List` and `MarkShipped` methods. I want to protect all of the action methods defined by the `Admin` controller, and I can do this by applying the `Authorize` attribute to the controller class, which then applies the authorization policy to all the action methods it contains, as shown in Listing 12-7.

Listing 12-7. Restricting Access in the AdminController.cs File

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;

namespace SportsStore.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;

        public AdminController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index() => View(repository.Products);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));

        [HttpPost]
        public IActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name} has been saved";
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the data values
                return View(product);
            }
        }
```

```
        public ViewResult Create() => View("Edit", new Product());

        [HttpPost]
        public IActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteProduct(productId);
            if (deletedProduct != null) {
                TempData["message"] = $"{deletedProduct.Name} was deleted";
            }
            return RedirectToAction("Index");
        }
    }
}
```

Creating the Account Controller and Views

When an unauthenticated user sends a request that requires authorization, they are redirected to the `/Account/Login` URL, which the application can use to prompt the user for their credentials. In preparation, I added a view model to represent the user's credentials by adding a class file called `LoginModel.cs` to the `Models/ViewModels` folder and using it to define the class shown in Listing 12-8.

Listing 12-8. The Contents of the LoginModel.cs File in the Models/ViewModels Folder

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels {

    public class LoginModel {

        [Required]
        public string Name { get; set; }

        [Required]
        [UIHint("password")]
        public string Password { get; set; }

        public string ReturnUrl { get; set; } = "/";
    }
}
```

The `Name` and `Password` properties have been decorated with the `Required` attribute, which uses model validation to ensure that values have been provided. The `Password` property has been decorated with the `UIHint` attribute so that when I use the `asp-for` attribute on the `input` element in the login Razor view, the tag helper will set the `type` attribute to `password`; that

way, the text entered by the user isn't visible onscreen. I describe the use of the `UIHint` attribute in Chapter 24.

Next, I added a class file called `AccountController.cs` to the `Controllers` folder and used it to define the controller shown in Listing 12-9. This is the controller that will respond to requests to the `/Account/Login` URL.

Listing 12-9. The Contents of the AccountController.cs File in the Controllers Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    [Authorize]
    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManager;

        public AccountController(UserManager<IdentityUser> userMgr,
                SignInManager<IdentityUser> signInMgr) {
            userManager = userMgr;
            signInManager = signInMgr;
        }

        [AllowAnonymous]
        public ViewResult Login(string returnUrl) {
            return View(new LoginModel {
                ReturnUrl = returnUrl
            });
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginModel loginModel) {
            if (ModelState.IsValid) {
                IdentityUser user =
                    await userManager.FindByNameAsync(loginModel.Name);
                if (user != null) {
                    await signInManager.SignOutAsync();
                    if ((await signInManager.PasswordSignInAsync(user,
                            loginModel.Password, false, false)).Succeeded) {
                        return Redirect(loginModel?.ReturnUrl ?? "/Admin/Index");
```

```
                    }
                }
            }
            ModelState.AddModelError("", "Invalid name or password");
            return View(loginModel);
        }

        public async Task<RedirectResult> Logout(string returnUrl = "/") {
            await signInManager.SignOutAsync();
            return Redirect(returnUrl);
        }
    }
}
```

When the user is redirected to the `/Account/Login` URL, the GET version of the `Login` action method renders the default view for the page, providing a view model object that includes the URL that the browser should be redirected to if the authentication request is successful.

Authentication credentials are submitted to the POST version of the `Login` method, which uses the `UserManager<IdentityUser>` and `SignInManager<IdentityUser>` services that have been received through the controller's constructor to authenticate the user and log them into the system. I explain how these classes work in Chapters 28–30, but for now it is enough to know that if there is an authentication failure, then I create a model validation error and render the default view; however, if authentication is successful, then I redirect the user to the URL that they want to access before they are prompted for their credentials.

Caution In general, using client-side data validation is a good idea. It offloads some of the work from your server and gives users immediate feedback about the data they are providing. However, you should not be tempted to perform authentication at the client, as this would typically involve sending valid credentials to the client so they can be used to check the username and password that the user has entered, or at least trusting the client's report of whether they have successfully authenticated. Authentication should always be done at the server.

To provide the `Login` method with a view to render, I created the `Views/Account` folder and added a Razor view file called `Login.cshtml` with the contents shown in Listing 12-10.

Listing 12-10. The Contents of the Login.cshtml File in the Views/Account Folder

```
@model LoginModel
```

```
@{
    ViewBag.Title = "Log In";
    Layout = "_AdminLayout";
}

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

<form asp-action="Login" asp-controller="Account" method="post">
    <input type="hidden" asp-for="ReturnUrl" />
    <div class="form-group">
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="text-danger"></span></div>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <div><span asp-validation-for="Password" class="text-danger"></span></div>
        <input asp-for="Password" class="form-control" />
    </div>
    <button class="btn btn-primary" type="submit">Log In</button>
</form>
```

The final step is a change to the shared administration layout to add a button that will log the current user out by sending a request to the `Logout` action, as shown in Listing 12-11. This is a useful feature that makes it easier to test the application, without which you would need to clear the browser's cookies in order to return to the unauthenticated state.

Listing 12-11. Adding a Logout Button in the _AdminLayout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
    <style>
        .input-validation-error { border-color: red; background-color: #fee ; }
    </style>
    <script asp-src-include="lib/jquery/**/jquery.min.js"></script>
    <script asp-src-include="lib/jquery-validation/**/jquery.validate.min.js">
    </script>
    <script asp-src-include="lib/jquery-validation-unobtrusive/**/*.min.js"></script>
</head>
<body class="panel panel-default">
    <div class="panel-heading">
        <h4>
            @ViewBag.Title
            <a class="btn btn-sm btn-primary pull-right"
```

```
                asp-action="Logout" asp-controller="Account">Log Out</a>
        </h4>
    </div>
    <div class="panel-body">
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>
```

Testing the Security Policy

Everything is in place and you can test the security policy by starting the application and requesting the `/Admin/Index` URL. Since you are presently unauthenticated and you are trying to target an action that requires authorization, your browser will be redirected to the `/Account/Login` URL. Enter **Admin** and **Secret123$** as the name and password and submit the form. The `Account` controller will check the credentials you provided with the seed data added to the Identity database and—assuming you entered the right details—authenticate you and redirect you back to the `/Account/Login` URL, to which you now have access. Figure 12-1 illustrates the process.

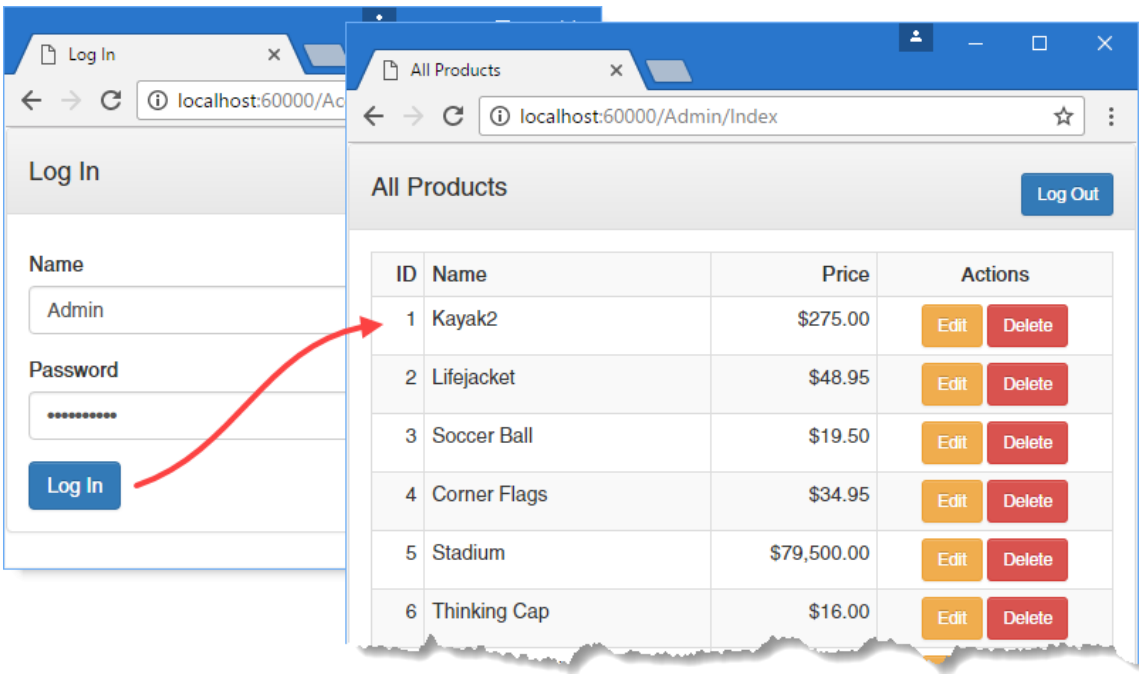

Figure 12-1. The administration authentication/authorization process

Summary

In this and previous chapters, I demonstrated how the ASP.NET Core MVC can be used to create a realistic e-commerce application. This extended example introduced many key MVC features: controllers, action methods, routing, views, metadata, validation, layouts, authentication, and more. You also saw how some of the key technologies related to MVC can be used. These included the Entity Framework Core, dependency injection, and unit testing. The result is an application that has a clean, component-oriented architecture that separates the various concerns and a code base that will be easy to extend and maintain. And that's the end of the SportsStore application. In the next chapter, I show you how to use Visual Studio Code to create ASP.NET Core MVC applications.