CHAPTER 11

SportsStore: Security and Deployment

Authentication and authorization are provided by the ASP.NET Core Identity system, which integrates neatly into the ASP.NET Core platform and the individual application frameworks. 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 more information in Chapters 37 and 38, where I show you how to create and manage user accounts and how to perform authorization using roles. But, as I noted previously, ASP.NET Core Identity is a large framework in its own right, and I cover only the basic features in this book.

My goal in this chapter 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 ASP.NET Core application.

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.

Installing the Identity Package for Entity Framework Core

To add the package that contains the ASP.NET Core Identity support for Entity Framework Core, use a PowerShell command prompt to run the command shown in Listing 11-1 in the `SportsStore` folder.

Listing 11-1. Installing the Entity Framework Core Package

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 6.0.0
```

Creating the Context Class

I need to create a database context file that will act 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 11-2.

Listing 11-2. The Contents of the AppIdentityDbContext.cs File in the SportsStore/Models Folder

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

namespace SportsStore.Models {

    public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {

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

The `AppIdentityDbContext` 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.

Defining the Connection String

The next step is to define the connection string that will be for the database. Listing 11-3 shows the addition of the connection string 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.

Listing 11-3. Defining a Connection String in the appsettings.json File in the SportsStore Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SportsStoreConnection": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;Multiple
    ActiveResultSets=true",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;MultipleActive
    ResultSets=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 IdentityConnection that specifies a LocalDB database called Identity.

Configuring the Application

Like other ASP.NET Core features, Identity is configured in the Program.cs file. Listing 11-4 shows the additions I made to set up Identity in the SportsStore project, using the context class and connection string defined previously.

Listing 11-4. Configuring Identity in the Program.cs File in the SportsStore Folder

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

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<StoreDbContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:SportsStoreConnection"]);
});

builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();
builder.Services.AddScoped<IOrderRepository, EFOrderRepository>();

builder.Services.AddRazorPages();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();
builder.Services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
builder.Services.AddServerSideBlazor();

builder.Services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration["ConnectionStrings:IdentityConnection"]));
builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AppIdentityDbContext>();

var app = builder.Build();

app.UseStaticFiles();
app.UseSession();

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

app.MapControllerRoute("catpage",
    "{category}/Page{productPage:int}",
    new { Controller = "Home", action = "Index" });
```

```
app.MapControllerRoute("page", "Page{productPage:int}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("category", "{category}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("pagination",
    "Products/Page{productPage}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapDefaultControllerRoute();
app.MapRazorPages();
app.MapBlazorHub();
app.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");

SeedData.EnsurePopulated(app);

app.Run();
```

In the listing, 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. I called the `UseAuthentication` and `UseAuthorization` methods to set up the middleware components that implement the security policy.

Creating and Applying the Database Migration

The basic configuration is 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 a new command prompt or PowerShell window and run the command shown in Listing 11-5 in the `SportsStore` folder to create a new migration for the Identity database.

Listing 11-5. Creating the Identity Migration

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

The important difference from previous database commands is that I have used the `-context` argument 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 context class.

Once Entity Framework Core has generated the initial migration, run the command shown in Listing 11-6 in the `SportsStore` folder to create the database and apply the migration.

Listing 11-6. Applying the Identity Migration

```
dotnet ef database update --context AppIdentityDbContext
```

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

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 11-7.

Listing 11-7. The Contents of the IdentitySeedData.cs File in the SportsStore/Models Folder

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

namespace SportsStore.Models {

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

        public static async void EnsurePopulated(IApplicationBuilder app) {

            AppIdentityDbContext context = app.ApplicationServices
                .CreateScope().ServiceProvider
                .GetRequiredService<AppIdentityDbContext>();
            if (context.Database.GetPendingMigrations().Any()) {
                context.Database.Migrate();
            }

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

            IdentityUser user = await userManager.FindByNameAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                user.Email = "admin@example.com";
                user.PhoneNumber = "555-1234";
                await userManager.CreateAsync(user, adminPassword);
            }
        }
    }
}
```

This code ensures the database is created and up-to-date and uses the UserManager<T> class, which is provided as a service by ASP.NET Core Identity for managing users, as described in Chapter 38. 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 38 for details of how to change the validation settings.

■ **Caution** Hard-coding the details of an administrator 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 38 for details of how to change passwords using Identity. See Chapter 15 for how to keep sensitive data, such as default passwords, out of source code control.

To ensure that the Identity database is seeded when the application starts, I added the statement shown in Listing 11-8 to the Configure method of the Program.cs file.

Listing 11-8. Seeding the Identity Database in the Program.cs File in the SportsStore Folder

```
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

using Microsoft.AspNetCore.Identity;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<StoreDbContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:SportsStoreConnection"]);
});

builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();
builder.Services.AddScoped<IOrderRepository, EFOrderRepository>();

builder.Services.AddRazorPages();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();
builder.Services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
builder.Services.AddServerSideBlazor();

builder.Services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration["ConnectionStrings:IdentityConnection"]));

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

var app = builder.Build();

app.UseStaticFiles();
app.UseSession();

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

app.MapControllerRoute("catpage",
    "{category}/Page{productPage:int}",
    new { Controller = "Home", action = "Index" });
```

```
app.MapControllerRoute("page", "Page{productPage:int}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("category", "{category}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("pagination",
    "Products/Page{productPage}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapDefaultControllerRoute();
app.MapRazorPages();
app.MapBlazorHub();
app.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");

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

app.Run();
```

DELETING AND RE-CREATING THE ASP.NET CORE IDENTITY DATABASE

If you need to reset the Identity database, then run the following command:

```
dotnet ef database drop --force --context AppIdentityDbContext
```

Restart the application, and the database will be re-created and populated with seed data.

Adding a Conventional Administration Feature

In Listing 11-9, I used Blazor to create the administration features so that I could demonstrate a wide range of ASP.NET Core features in the SportsStore project. Although Blazor is useful, it is not suitable for all projects—as I explain in Part 4—and most projects are likely to use controllers or Razor Pages for their administration features. I describe the way that ASP.NET Core Identity works with all the application frameworks in Chapter 38, but just to provide a balance to the all-Blazor tools created in Chapter 10, I am going to create a Razor Page that will display the list of users in the ASP.NET Core Identity database. I describe how to manage the Identity database in more detail in Chapter 38, and this Razor Page is just to add a sensitive feature to the SportsStore application that isn't created with Blazor. Add a Razor Page named `IdentityUsers.cshtml` to the `SportsStore/Pages/Admin` folder with the contents shown in Listing 11-9.

Listing 11-9. The Contents of the IdentityUsers.cshtml File in the SportsStore/Pages/Admin Folder

```
@page
@model IdentityUsersModel
@using Microsoft.AspNetCore.Identity

<h3 class="bg-primary text-white text-center p-2">Admin User</h3>
```

```
<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>User</th><td>@Model.AdminUser.UserName</td></tr>
        <tr><th>Email</th><td>@Model.AdminUser.Email</td></tr>
        <tr><th>Phone</th><td>@Model.AdminUser.PhoneNumber</td></tr>
    </tbody>
</table>

@functions{

    public class IdentityUsersModel: PageModel {
        private UserManager<IdentityUser> userManager;

        public IdentityUsersModel(UserManager<IdentityUser> mgr) {
            userManager = mgr;
        }

        public IdentityUser AdminUser { get; set; } = new();

        public async Task OnGetAsync() {
            AdminUser = await userManager.FindByNameAsync("Admin");
        }
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/admin/identityusers` to see the content generated by the Razor Page, which is shown in Figure 11-1.

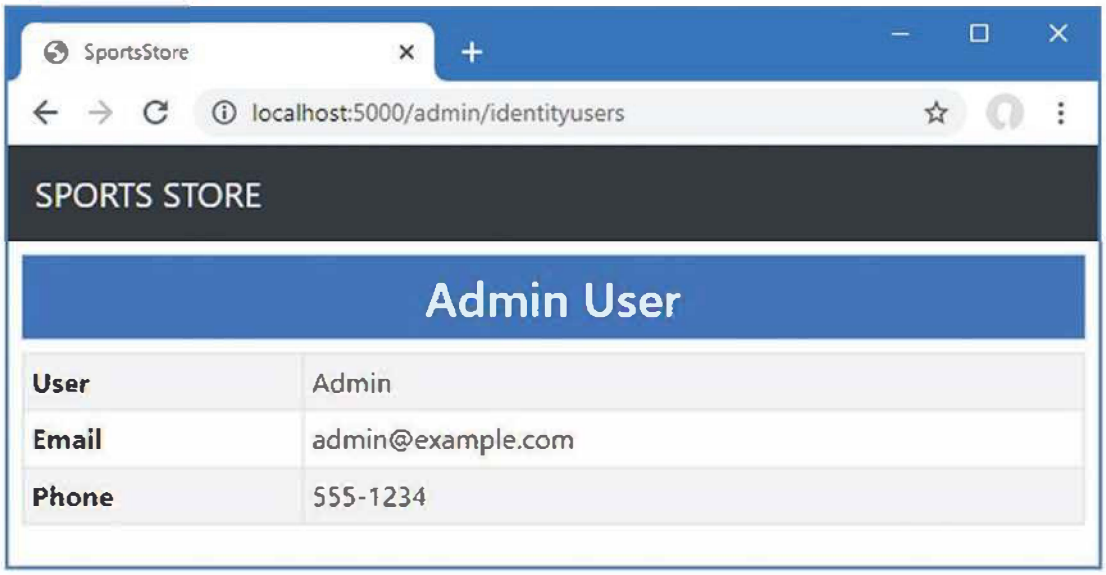

Figure 11-1. *A Razor Page administration feature*

Applying a Basic Authorization Policy

Now that I have 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 37 and 38, but since the SportsStore application has only one user, distinguishing between anonymous and authenticated requests is sufficient.

For controllers and Razor pages, the `Authorize` attribute is used to restrict access, as shown in Listing 11-10.

Listing 11-10. Restricting Access in the IdentityUsers.cshtml File in the SportsStore/Pages/Admin Folder

```
@page
@model IdentityUsersModel
@using Microsoft.AspNetCore.Identity
@using Microsoft.AspNetCore.Authorization

<h3 class="bg-primary text-white text-center p-2">Admin User</h3>

<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>User</th><td>@Model.AdminUser.UserName</td></tr>
        <tr><th>Email</th><td>@Model.AdminUser.Email</td></tr>
        <tr><th>Phone</th><td>@Model.AdminUser.PhoneNumber</td></tr>
    </tbody>
</table>

@functions{

    [Authorize]
    public class IdentityUsersModel: PageModel {
        private UserManager<IdentityUser> userManager;

        public IdentityUsersModel(UserManager<IdentityUser> mgr) {
            userManager = mgr;
        }

        public IdentityUser AdminUser { get; set; } = new();

        public async Task OnGetAsync() {
            AdminUser = await userManager.FindByNameAsync("Admin");
        }
    }
}
```

When there are only authorized and unauthorized users, the `Authorize` attribute can be applied to the Razor Page that acts as the entry point for the Blazor part of the application, as shown in Listing 11-11.

Listing 11-11. Applying Authorization in the Index.cshtml File in the SportsStore/Pages/Admin Folder

```
@page "/admin"
@{  Layout = null; }
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]

<!DOCTYPE html>
<html>
<head>
    <title>SportsStore Admin</title>
    <link href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="/" />
</head>
<body>
    <component type="typeof(Routed)" render-mode="Server" />
    <script src="/_framework/blazor.server.js"></script>
</body>
</html>
```

Since this Razor Page has been configured with a page model class, I can apply the attribute with an `@attribute` expression.

Creating the Account Controller and Views

When an unauthenticated user sends a request that requires authorization, the user is redirected to the `/Account/Login` URL, which the application can use to prompt the user for their credentials. In Chapters 38 and 39, I show you how to handle authentication using Razor Pages, so, for variety, I am going to use controllers and views for SportsStore. 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 11-12.

Listing 11-12. The Contents of the LoginModel.cs File in the SportsStore/Models/ViewModels Folder

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels {

    public class LoginModel {

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

        [Required]
        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. Next, I added a class file called AccountController.cs to the Controllers folder and used it to define the controller shown in Listing 11-13. This is the controller that will respond to requests to the /Account/Login URL.

Listing 11-13. The Contents of the AccountController.cs File in the SportsStore/Controllers Folder

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

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

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

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

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

```
        [Authorize]
        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 to which the browser should be redirected 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 37 and 38, 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 11-14.

Listing 11-14. The Contents of the Login.cshtml File in the SportsStore/Views/Account Folder

```
@model LoginModel
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
    <div class="m-1 p-1">
        <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 asp-validation-for="Name" class="text-danger"></div>
                <input asp-for="Name" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="Password"></label>
                <div asp-validation-for="Password" class="text-danger"></div>
                <input asp-for="Password" type="password" class="form-control" />
            </div>
            <button class="btn btn-primary mt-2" type="submit">Log In</button>
        </form>
    </div>
</body>
</html>
```

The final step is a change to the shared administration layout to add a button that will log out the current user by sending a request to the Logout action, as shown in Listing 11-15. This is a useful feature that makes it easier to test the application, without which you would need to clear the browser's cookies to return to the unauthenticated state.

Listing 11-15. Adding a Logout Button in the AdminLayout.razor File in the SportsStore/Pages/Admin Folder

```
@inherits LayoutComponentBase

<div class="bg-info text-white p-2">
    <div class="container-fluid">
        <div class="row">
            <div class="col">
                <span class="navbar-brand ml-2">SPORTS STORE Administration</span>
            </div>
            <div class="col-2 text-right">
                <a class="btn btn-sm btn-primary" href="/account/logout">Log Out</a>
            </div>
        </div>
    </div>
</div>
<div class="container-fluid">
    <div class="row p-2">
        <div class="col-3">
            <div class="d-grid gap-1">
                <NavLink class="btn btn-outline-primary"
                         href="/admin/products"
                         ActiveClass="btn-primary text-white"
                         Match="NavLinkMatch.Prefix">
                    Products
                </NavLink>
                <NavLink class="btn btn-outline-primary"
                         href="/admin/orders"
                         ActiveClass="btn-primary text-white"
                         Match="NavLinkMatch.Prefix">
```

```
                    Orders
                </NavLink>
            </div>
        </div>
        <div class="col">
            @Body
        </div>
    </div>
</div>
```

Testing the Security Policy

Everything is in place, and you can test the security policy by restarting ASP.NET Core and requesting `http://localhost:5000/admin` or `http://localhost:5000/admin/identityusers`.

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 to the URL you requested, to which you now have access. Figure 11-2 illustrates the process.

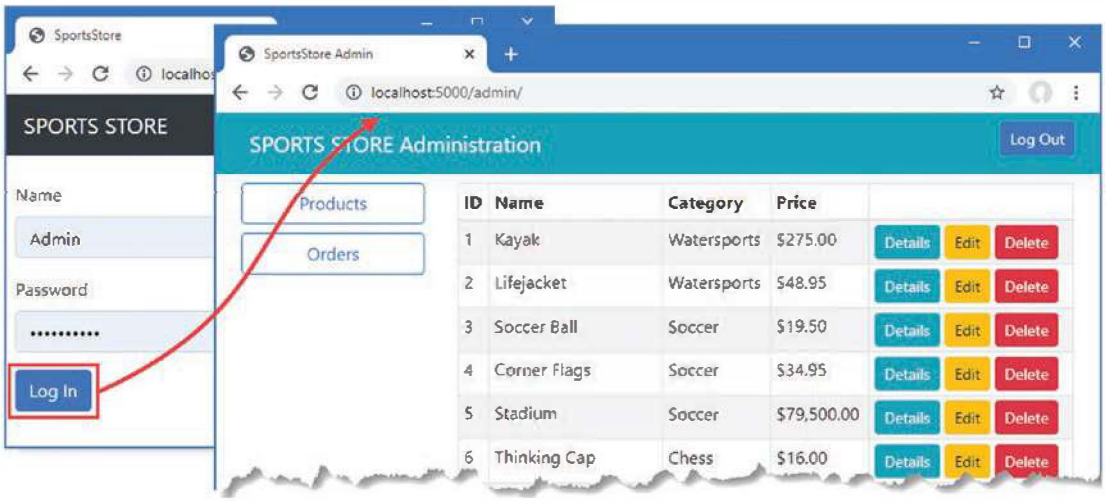

Figure 11-2. *The administration authentication/authorization process*

Preparing ASP.NET Core for Deployment

In this section, I will prepare SportsStore and create a container that can be deployed into production. There is a wide range of deployment models available for ASP.NET Core applications, but I have picked Docker containers because they can be run on most hosting platforms or be deployed into a private data center. This is not a complete guide to deployment, but it will give you a sense of the process to prepare an application.

Configuring Error Handling

At the moment, the application is configured to use the developer-friendly error pages, which provide helpful information when a problem occurs. This is not information that end users should see, so I added a Razor Page named `Error.cshtml` to the `Pages` folder with the content shown in Listing 11-16.

Listing 11-16. The Contents of the Error.cshtml File in the Pages Folder

```
@page "/error"
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <title>Error</title>
</head>
<body class="text-center">
    <h2 class="text-danger">Error.</h2>
    <h3 class="text-danger">An error occurred while processing your request</h3>
</body>
</html>
```

This kind of error page is the last resort, and it is best to keep it as simple as possible and not to rely on shared views, view components, or other rich features. In this case, I have disabled shared layouts and defined a simple HTML document that explains that there has been an error, without providing any information about what has happened.

In Listing 11-17, I have reconfigured the application so that the `Error` page is used for unhandled exceptions when the application is in the production environment. I have also set the local, which is required when deploying to a Docker container. The local I have chosen is `en-US`, which represents the language and currency conventions of English as it is spoken in the United States.

Listing 11-17. Configuring Error Handling in the Program.cs File in the SportsStore Folder

```
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

using Microsoft.AspNetCore.Identity;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<StoreDbContext>(opts => {
    opts.UseSqlServer(
        builder.Configuration["ConnectionStrings:SportsStoreConnection"]);
});

builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();
builder.Services.AddScoped<IOrderRepository, EFOrderRepository>();
```

```
builder.Services.AddRazorPages();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession();
builder.Services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
builder.Services.AddServerSideBlazor();

builder.Services.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration["ConnectionStrings:IdentityConnection"]));

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

var app = builder.Build();

if (app.Environment.IsProduction()) {
    app.UseExceptionHandler("/error");
}

app.UseRequestLocalization(opts => {
    opts.AddSupportedCultures("en-US")
    .AddSupportedUICultures("en-US")
    .SetDefaultCulture("en-US");
});

app.UseStaticFiles();
app.UseSession();

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

app.MapControllerRoute("catpage",
    "{category}/Page{productPage:int}",
    new { Controller = "Home", action = "Index" });

app.MapControllerRoute("page", "Page{productPage:int}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("category", "{category}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapControllerRoute("pagination",
    "Products/Page{productPage}",
    new { Controller = "Home", action = "Index", productPage = 1 });

app.MapDefaultControllerRoute();
app.MapRazorPages();
app.MapBlazorHub();
app.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");
```

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

app.Run();
```

As I explain in Chapter 12, the `IWebHostEnvironment` interface describes the environment in which the application is running. The changes mean that the `UseExceptionHandler` method is called when the application is in production, but the developer-friendly error pages are used otherwise.

Creating the Production Configuration Settings

The JSON configuration files that are used to define settings such as connection strings can be created so they apply only when the application is in a specific environment, such as development, staging, or production. The template I used to create the SportsStore project in Chapter 7 created the `appsettings.json` and `appsettings.Development.json` files, which are intended to be the default settings that are overridden with those that are specific for development. I am going to take the reverse approach for this chapter and define a file that contains just those settings that are specific to production. Add a JSON file named `appsettings.Production.json` to the `SportsStore` folder with the content shown in Listing 11-18.

■ **Caution** Do not use these connection strings in real projects. You must correctly describe the connection to your production database, which is unlikely to be the same as the ones in the listing.

Listing 11-18. The Contents of the appsettings.Production.json File in the SportsStore Folder

```
{
  "ConnectionStrings": {
    "SportsStoreConnection": "Server=sqlserver;Database=SportsStore;MultipleActiveResult
    Sets=true;User=sa;Password=MyDatabaseSecret123",
    "IdentityConnection": "Server=sqlserver;Database=Identity;MultipleActiveResultSets=true;
    User=sa;Password=MyDatabaseSecret123"
  }
}
```

These connection strings, each of which is defined on a single line, describe connections to SQL Server running on `sqlserver`, which is another Docker container running SQL Server.

Creating the Docker Image

In the sections that follow, I configure and create the Docker image for the application that can be deployed into a container environment such as Microsoft Azure or Amazon Web Services. Bear in mind that containers are only one style of deployment and there are many others available if this approach does not suit you.

■ **Note** Bear in mind that I am going to connect to a database running on the development machine, which is not how most real applications are configured. Be sure to configure the database connection strings and the container networking settings to match your production environment.

Installing Docker Desktop

Go to Docker.com and download and install the Docker Desktop package. Follow the installation process, reboot your Windows machine, and run the command shown in Listing 11-19 to check that Docker has been installed and is in your path. (The Docker installation process seems to change often, which is why I have not been more specific about the process.)

■ **Note** You will have to create an account on Docker.com to download the installer.

Listing 11-19. Checking the Docker Desktop Installation

```
docker --version
```

Creating the Docker Configuration Files

Docker is configured using a file named `Dockerfile`. There is no Visual Studio item template for this file, so use the Text File template to add a file named `Dockerfile.text` to the project and then rename the file to `Dockerfile`. If you are using Visual Studio Code, you can just create a file named `Dockerfile` without the extension. Use the configuration settings shown in Listing 11-20 as the contents for the new file.

Listing 11-20. The Contents of the Dockerfile File in the SportsStore Folder

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0

COPY /bin/Release/net6.0/publish/ SportsStore/

ENV ASPNETCORE_ENVIRONMENT Production
ENV Logging__Console__FormatterName=Simple

EXPOSE 5000
WORKDIR /SportsStore
ENTRYPOINT ["dotnet", "SportsStore.dll",  "--urls=http://0.0.0.0:5000"]
```

These instructions copy the SportsStore application into a Docker image and configure its execution. Next, create a file called `docker-compose.yml` with the content shown in Listing 11-21. Visual Studio doesn't have a template for this type of file, but if you select the Text File template and enter the complete file name, it will create the file. Visual Studio Code users can simply create a file named `docker-compose.yml`.

Listing 11-21. The Contents of the docker-compose.yml File in the SportsStore Folder

```
version: "3"
services:
    sportsstore:
        build: .
        ports:
            - "5000:5000"
```

```
    environment:
        - ASPNETCORE_ENVIRONMENT=Production
    depends_on:
        - sqlserver
sqlserver:
    image: "mcr.microsoft.com/mssql/server"
    environment:
        SA_PASSWORD: "MyDatabaseSecret123"
        ACCEPT_EULA: "Y"
```

The YML files are especially sensitive to formatting and indentation, and it is important to create this file exactly as shown. If you have problems, then use the `docker-compose.yml` file from the GitHub repository for this book, https://github.com/apress/pro-asp.net-core-3.

Publishing and Imaging the Application

Prepare the SportsStore application by using a PowerShell prompt to run the command shown Listing 11-22 in the `SportsStore` folder.

Listing 11-22. Preparing the Application

```
dotnet publish -c Release
```

Next, run the command shown in Listing 11-23 to create the Docker image for the SportsStore application. This command will take some time to complete the first time it is run because it will download the Docker images for ASP.NET Core.

Listing 11-23. Performing the Docker Build

```
docker-compose build
```

The first time you run this command, you may be prompted to allow Docker to use the network, as shown in Figure 11-3.

Figure 11-3. *Granting network access*

Click the Allow button, return to the PowerShell prompt, use Control+C to terminate the Docker containers, and run the command in Listing 11-23 again.

Running the Containerized Application

Run the command shown in Listing 11-24 in the `SportsStore` folder to start the Docker containers for SQL Server.

Listing 11-24. Starting the Database Container

```
docker-compose up sqlserver
```

This command will take some time to complete the first time it is run because it will download the Docker images for SQL Server. You will see a large amount of output as SQL Server starts up. Once the database is running, use a separate command prompt to run the command shown in Listing 11-25 to start the container for the SportsStore application.

Listing 11-25. Starting the SportsStore Container

```
docker-compose up sportsstore
```

The application will be ready when you see output like this:

```
...
sportsstore_1  | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1  |       Now listening on: http://0.0.0.0:5000
sportsstore_1  | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1  |       Application started. Press Ctrl+C to shut down.
sportsstore_1  | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1  |       Hosting environment: Production
sportsstore_1  | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1  |       Content root path: /SportsStore
...
```

Open a new browser window and request `http://localhost:5000`, and you will receive a response from the containerized version of SportsStore, as shown in Figure 11-4, which is now ready for deployment. Use Control+C at the PowerShell command prompts to terminate the Docker containers.

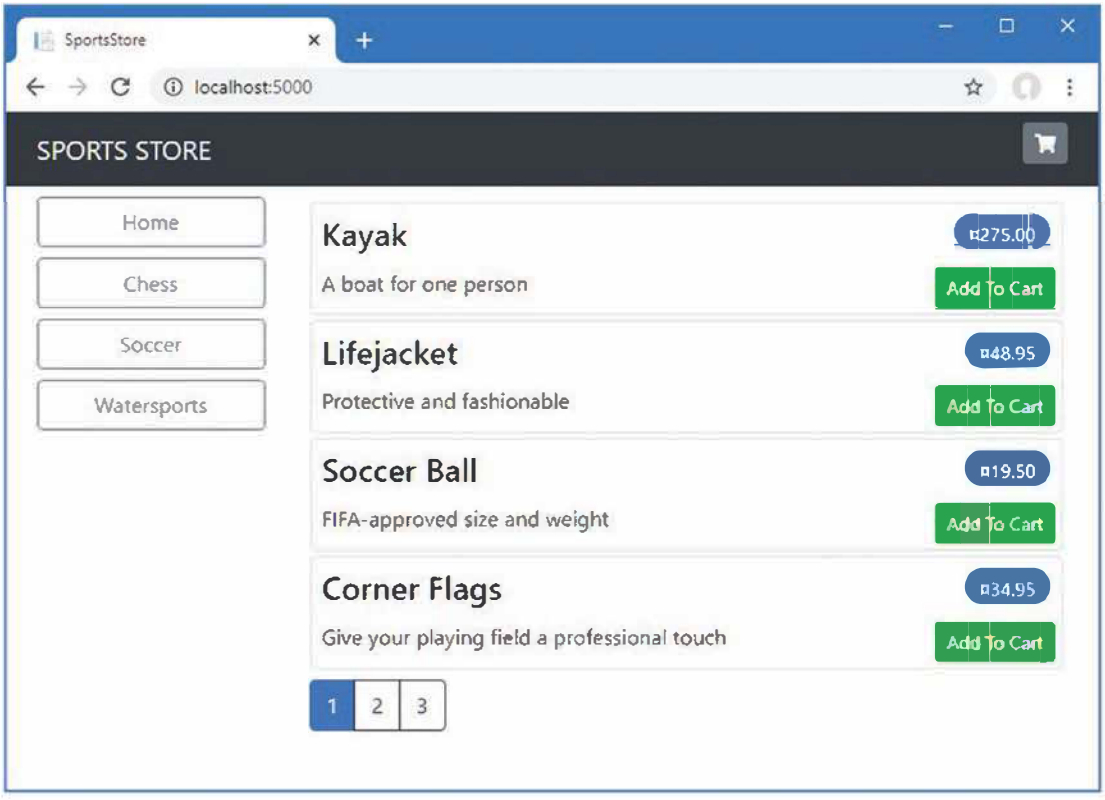

Figure 11-4. *Running the SportsStore application in a container*

Summary

In this and previous chapters, I demonstrated how ASP.NET Core can be used to create a realistic e-commerce application. This extended example introduced many key features: controllers, action methods, views, Razor Pages, Blazor, routing, validation, authentication, and more. You also saw how some of the key technologies related to how ASP.NET Core can be used. These included the Entity Framework Core, ASP.NET Core Identity, and unit testing. And that's the end of the SportsStore application. In the next part of the book, I start to dig into the details of ASP.NET Core.