CHAPTER 12

SportsStore: Security and Deployment

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. Once I have the security in place, I will show you how to prepare and deploy the SportsStore application into production.

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 more detailed information in Chapters 28, 29, and 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.

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 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 12-1.

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_12

■ **Note** You might be used to adding packages to the project to get additional features like security working. But, with the release of ASP.NET Core 2, the NuGet packages required for Identity are already included in the project through the meta-package that was added to the `SportsStore.csproj` file as part of the project template.

Listing 12-1. The Contents of the AppIdentityDbContext.cs File in the 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. 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 12-2, 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 12-2. Defining a Connection String in the appsettings.json File

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;Trusted_
                          Connection=True;MultipleActiveResultSets=true"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;Trusted_
                          Connection=True;MultipleActiveResultSets=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 12-3 shows the additions I made to set up Identity in the SportsStore project, using the context class and connection string defined previously.

Listing 12-3. Configuring Identity in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        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>()
                .AddDefaultTokenProviders();

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

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

            });
            SeedData.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 `UseAuthentication` method to set up the components that will intercept requests and responses to 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 following command in the SportsStore project folder to create a new migration for the Identity database:

```
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 following command to create the database and run the migration commands:

```
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 12-4.

Listing 12-4. The Contents of the IdentitySeedData.cs File in the Models Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
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 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 28 for details of how to change passwords using Identity.

To ensure that the Identity database is seeded when the application starts, I added the statement shown in Listing 12-5 to the `Configure` method of the `Startup` class.

Listing 12-5. Seeding the Identity Database in the Startup.cs File in the SportsStore Folder

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseMvc(routes => {
```

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

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

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 28,29, and 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 administrative 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 on-screen. 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.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, 29, and 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 src="/lib/jquery/dist/jquery.min.js"></script>
    <script src="/lib/jquery-validation/dist/jquery.validate.min.js"></script>
    <script
      src="/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
    </script>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2 row">
        <div class="col">
            <h4>@ViewBag.Title</h4>
        </div>
        <div class="col-2">
            <a class="btn btn-sm btn-primary"
               asp-action="Logout" asp-controller="Account">Log Out</a>
        </div>
    </div>
    @if (TempData["message"] != null) {
        <div class="alert alert-success mt-1">@TempData["message"]</div>
    }
    @RenderBody()
</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.

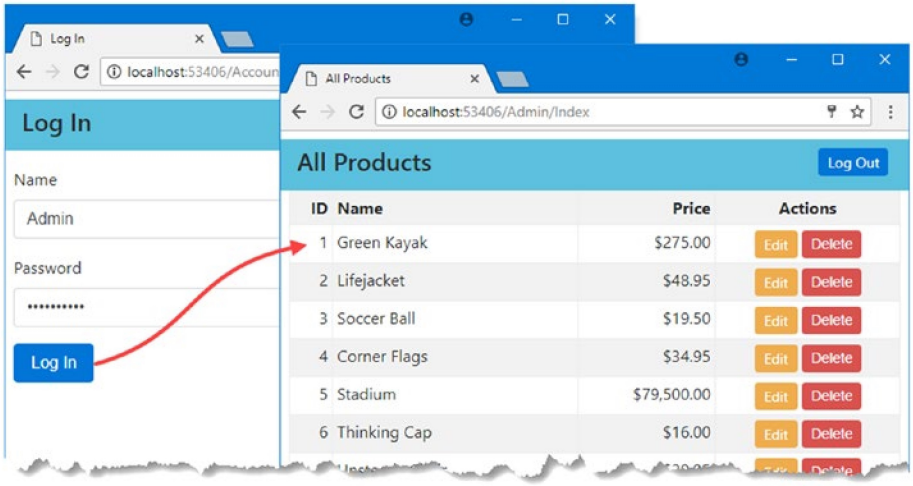

Figure 12-1. *The administration authentication/authorization process*

Deploying the Application

All the features and functionality for the SportsStore application are in place, so it is time to prepare the application and deploy it into production. Lots of hosting options are available for ASP.NET Core MVC applications, and the one that I use in this chapter is the Microsoft Azure platform, which I have chosen because it comes from Microsoft and because it offers free accounts, which means you can follow the SportsStore example all the way through, even if you don't want to use Azure for your own projects.

■ **Note** You will need an Azure account for this section. If you don't have one, you can create a free account at `http://azure.microsoft.com`.

Creating the Databases

The starting point is to create the databases that the SportsStore application will use in production. This is something that you can do as part of the Visual Studio deployment process, but it is a chicken-and-egg situation because you need to know the connection strings for the databases before you deploy, which is the process that creates the databases.

■ **Caution** The Azure portal changes often as Microsoft adds new features and revises existing ones. The instructions in this section were accurate when I wrote them, but the required steps may have changed slightly by the time you read this. The basic approach should still be the same, but the names of data fields and the exact order of steps may require some experimentation to get the right results.

The simplest approach is to log in to `http://portal.azure.com` using your Azure account and create the databases manually. Once you are logged in, select the SQL Databases resource category and click the Add button to create a new database.

For the first database enter the name **products**. Click the Configure Required Settings link and then the Create a New Server link. Enter a new server name—which must be unique across Azure—and select a database administrator username and password. I entered the server name **sportsstorecore2db**, with the administrator name of **sportsstoreadmin** and a password of **Secret123$**. You will have to use a different server name, and I suggest that you use a more robust password. Select a location for your database; click the Select button to close the options and then the Create button to create the database itself. Azure will take a few minutes to perform the creation process, after which it will appear in the SQL Databases resource category.

Create another SQL server, this time entering the name **identity**. You can use the database server that you created a moment ago, rather than creating a new one. The result is two SQL Server databases hosted by Azure with the details shown in Table 12-1. You will have a different database server name and—ideally—better passwords.

Table 12-1. *The Azure Databases for the SportsStore Application*

Database Name	Server Name	Administrator	Password
products	sportsstorecore2db	sportsstoreadmin	Secret123$
identity	sportsstorecore2db	sportsstoreadmin	Secret123$

Opening Firewall Access for Configuration

I need to populate the databases with their schemas, and the simplest way to do that is by opening Azure firewall access so that I can run the Entity Framework Core commands from my development machine.

Select either of the databases in the SQL Databases resource category, click the Tools button, and then click the Open in Visual Studio link. Now click the Configure Your Firewall link, click the Add Client IP button, and then click Save. This allows your current IP address to reach the database server and perform configuration commands. (You can inspect the database schema by clicking the Open In Visual Studio button, which will open Visual Studio and use the SQL Server Object Explorer to examine the database.)

Getting the Connection Strings

I will need the connection strings for the new database shortly. Azure provides this information when you click a database in the SQL Databases resource category through a Show Database Connection Strings link. Connection strings are provided for different development platforms, and it is the ADO.NET strings that are required for .NET applications. Here is the connection string that the Azure portal provides for the `products` database:

```
Server=tcp:sportsstorecore2db.database.windows.net,1433;Initial Catalog=products;Persist
Security Info=False;User ID={your_username};Password={your_password};MultipleActiveResultS
ets=True;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
```

You will see different configuration options depending on how Azure provisioned your database. Notice that there are placeholders for the username and password, which I have marked in bold, that must be changed when you use the connection string to configure the application.

Preparing the Application

I have some basic preparation to do before I can deploy the application, to make it ready for the production environment. In the sections that follow, I change the way that errors are displayed and set up the production connection strings for the databases.

Creating the Error Controller and View

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 class file called `ErrorController.cs` to the `Controllers` folder and used it to define the simple controller shown in Listing 12-12.

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

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {

    public class ErrorController : Controller {

        public ViewResult Error() => View();
    }
}
```

The controller defines an `Error` action that renders the default view. To provide the controller with the view, I created the `Views/Error` folder, added a Razor view file called `Error.cshtml`, and applied the markup shown in Listing 12-13.

Listing 12-13. The Contents of the Error.cshtml File in the Views/Error Folder

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <title>Error</title>
</head>
<body>
    <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.

Defining the Production Database Settings

The next step is to create a file that will provide the application with its database connection strings in production. I added a new ASP.NET Configuration File called `appsettings.production.json` to the SportsStore project and added the content shown in Listing 12-14.

■ **Tip** The Solution Explorer nests this file inside `appsettings.json` in the file listing, which you will have to expand if you want to edit the file again later.

Listing 12-14. The Contents of the appsettings.production.json File

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=tcp:sportsstorecore2db.database.windows.net,1433;Initial
      Catalog=products;Persist Security Info=False;User ID={your_username};Password={your_
      password};MultipleActiveResultSets=True;Encrypt=True;TrustServerCertificate=False;
      Connection Timeout=30;"

    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=tcp:sportsstorecore2db.database.windows.net,1433;Initial
      Catalog=identity;Persist Security Info=False;User ID={your_username};Password={your_
      password};MultipleActiveResultSets=True;Encrypt=True;TrustServerCertificate=False;
      Connection Timeout=30;"
    }
  }
}
```

This file is hard to read because connection strings cannot be split across multiple lines. The contents of this file duplicate the connection strings section of the `appsettings.json` file but use the Azure connection strings. (Remember to replace the username and password placeholders.) I have also set the `MultipleActiveResultSets` to `True`, which allows multiple concurrent queries and avoids a common error condition that arises when performing complex LINQ queries of application data.

■ **Note** Remove the brace characters when you insert your username and password into the connection strings so that you end up with `Password=MyPassword` and not `Password={MyPassword}`.

Configuring the Application

Now I can change the Startup class so that the application behaves differently when in production. Listing 12-15 shows the changes I made.

Listing 12-15. Configuring the Application in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        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>()
                .AddDefaultTokenProviders();

            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) {

            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
                app.UseStatusCodePages();
            } else {
                app.UseExceptionHandler("/Error");
            }

            app.UseStaticFiles();
            app.UseSession();
            app.UseAuthentication();
            app.UseMvc(routes => {
                routes.MapRoute(name: "Error", template: "Error",
                    defaults: new { controller = "Error", action = "Error" });
                routes.MapRoute(name: null,
                    template: "{category}/Page{productPage:int}",
                    defaults: new { controller = "Product", action = "List" }
                );
                routes.MapRoute(name: null,template: "Page{productPage:int}",
                    defaults: new { controller = "Product",
                    action = "List", productPage = 1 }
                );
                routes.MapRoute(name: null, template: "{category}",
                    defaults: new { controller = "Product",
                        action = "List", productPage = 1 }
                );
                routes.MapRoute(name: null,template: "",
                    defaults: new { controller = "Product",
                        action = "List", productPage = 1 });
                routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
            });
            //SeedData.EnsurePopulated(app);
            //IdentitySeedData.EnsurePopulated(app);
        }
    }
}
```

The `IHostingEnvironment` interface is used to provide information about the environment in which the application is running, such as development or production. When the hosting environment is set to Production, then ASP.NET Core will load the `appsettings.production.json` file and its contents to override the settings in the `appsettings.json` file, which means that the Entity Framework Core will connect to the Azure databases instead of LocalDB. There are a lot of options available for tailoring the configuration of an application in different environments, which I explain in Chapter 14.

I have also commented out the statements that seed the databases, which I explain in the "Managing Database Seeding" section.

Applying the Database Migrations

To set up the databases with the schemas required for the application, open a new command prompt or PowerShell window and navigate to the SportsStore project directory. Setting the environment so that the `dotnet` command-line tool will use the connection strings for Azure requires setting an environment variable. If you are using PowerShell, use this command to set the environment variable:

```
$env:ASPNETCORE_ENVIRONMENT="Production"
```

If you are using a command prompt, then use this command to set the environment variable instead:

```
set ASPNETCORE_ENVIRONMENT=Production
```

Run the following commands in the SportsStore project folder to apply the migrations in the project to the Azure databases:

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

The environment variable specifies the hosting environment that is used to obtain the connection strings to reach the databases. If these commands do not work, ensure that you have configured the Azure firewall to allow access to your development machine, as described earlier in this chapter, and that you have correctly copied and modified the connection strings.

Managing Database Seeding

In Listing 12-15, I commented out the statements in the `Startup` class that seeded the databases. I did this because the Entity Framework Core commands used in the previous section to apply the migrations to the database rely on the services set up by the `Startup` class, which means that, with those statements enabled, the code to seed the databases would have been called before the migrations were applied, which would have resulted in an error and prevented the migrations from working. This didn't cause a problem when the databases were set up. For the `products` database, this was because the `SeedData.EnsurePopulated` method applies the migrations before seeding the data and because I didn't add the Identity seed data to the application until after I had applied the migration to the database.

For the production environment, I want to take a different approach to seed data. For the user accounts, I am going to populate the database with the administrator account when there is a login attempt. I am going to add a feature to the administration tool for seeding the product database so that the production system can be populated with data for testing data or left empty for real data as required.

■ **Note** Seeding authentication data in a production system should be done with care, and your application should use the features described in Chapters 28, 29, and 30 to change the password as soon as the application is deployed.

Seeding Identity Data

The first step in changing the way that user data is seeded is to simplify the code in the IdentitySeedData class, as shown in Listing 12-16.

Listing 12-16. Simplifying Code in the IdentitySeedData.cs File in the Models Folder

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

namespace SportsStore.Models {

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

        public static async Task EnsurePopulated(UserManager<IdentityUser>
                userManager) {

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

Rather than obtaining the UserManager<IdentityUser> service itself, the EnsurePopulated method receives the object as an argument. This allows me to integrate the database seeding in the AccountController class, as shown in Listing 12-17.

Listing 12-17. Seeding Data in the AccountController.cs File in the Controllers Folder

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

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;
```

```
            IdentitySeedData.EnsurePopulated(userMgr).Wait();
        }

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

These changes will ensure that the Identity database is seeded every time that an `AccountController` object is created to handle an HTTP request. This is not ideal, of course, but there is no good way to seed a database, and this approach will ensure that the application can be administered both in production and development, albeit at the cost of some additional database queries.

Seeding the Product Data

For the product data, I am going to present the administrator with a button that will seed the database when it is empty. The first step is to change the seeding code so that it uses an interface that will allow it to access services provided through a controller, rather than through the `Startup` class, as shown in Listing 12-18. I have also commented out the statement that automatically applies any pending migrations, which can cause data loss and should be used only with the greatest care in production systems.

Listing 12-18. Preparing for Manual Seeding in the SeedData.cs File in the Models Folder

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using System;

namespace SportsStore.Models {

    public static class SeedData {

        public static void EnsurePopulated(IServiceProvider services) {
            ApplicationDbContext context =
                services.GetRequiredService<ApplicationDbContext>();
            //context.Database.Migrate();
            if (!context.Products.Any()) {
                context.Products.AddRange(

                    // ...statements omiited for brevity...

                );
                context.SaveChanges();
            }
        }
    }
}
```

The next step is to update the `Admin` controller to add an action method that will trigger the seeding operation, as shown in Listing 12-19.

Listing 12-19. Seeding the Database in the AdminController.cs File in the Controllers Folder

```
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);

        // ...other methods omitted for brevity...

        [HttpPost]
        public IActionResult SeedDatabase() {
            SeedData.EnsurePopulated(HttpContext.RequestServices);
            return RedirectToAction(nameof(Index));
        }
    }
}
```

The new action is decorated with the `HttpPost` attribute so that it can be targeted with POST requests, and it will redirect the browser to the `Index` action method once the database has been seeded. All that remains is to create a button to seed the database that will be displayed when it is empty, as shown in Listing 12-20.

Listing 12-20. Adding a Button in the Index.cshtml File in the Views/Admin Folder

```
@model IEnumerable<Product>

@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}

@if (Model.Count() == 0) {
    <div class="text-center m-2">
        <form asp-action="SeedDatabase" method="post">
            <button type="submit" class="btn btn-danger">Seed Database</button>
        </form>
    </div>
} else {
    <table class="table table-striped table-bordered table-sm">
        <tr>
            <th class="text-right">ID</th>
```

```
            <th>Name</th>
            <th class="text-right">Price</th>
            <th class="text-center">Actions</th>
        </tr>
        @foreach (var item in Model) {
            <tr>
                <td class="text-right">@item.ProductID</td>
                <td>@item.Name</td>
                <td class="text-right">@item.Price.ToString("c")</td>
                <td class="text-center">
                    <form asp-action="Delete" method="post">
                        <a asp-action="Edit" class="btn btn-sm btn-warning"
                           asp-route-productId="@item.ProductID">
                            Edit
                        </a>
                        <input type="hidden" name="ProductID"
                           value="@item.ProductID" />
                        <button type="submit" class="btn btn-danger btn-sm">
                            Delete
                        </button>
                    </form>
                </td>
            </tr>
        }
    </table>
}
<div class="text-center">
    <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>
```

Deploying the Application

To deploy the application, right-click the SportsStore project in the Solution Explorer (the project, not the solution) and select Publish from the pop-up menu. Visual Studio will present you with a choice of publishing methods, as shown in Figure 12-2.

WHERE TO START IF DEPLOYMENT FAILS

The single biggest cause of failed deployments is connection strings, either because they were not copied correctly from Azure or because they were edited incorrectly to insert the username and password. If your deployment fails, then the connection strings are the place to start. If you don't get the expected results from the `dotnet ef database update` commands in the "Applying the Database Migrations" sections, then your deployment will fail. If the commands do work but deployment fails, then make sure you have set the environment variable because it is possible that you are preparing the local database and not the one in the cloud.

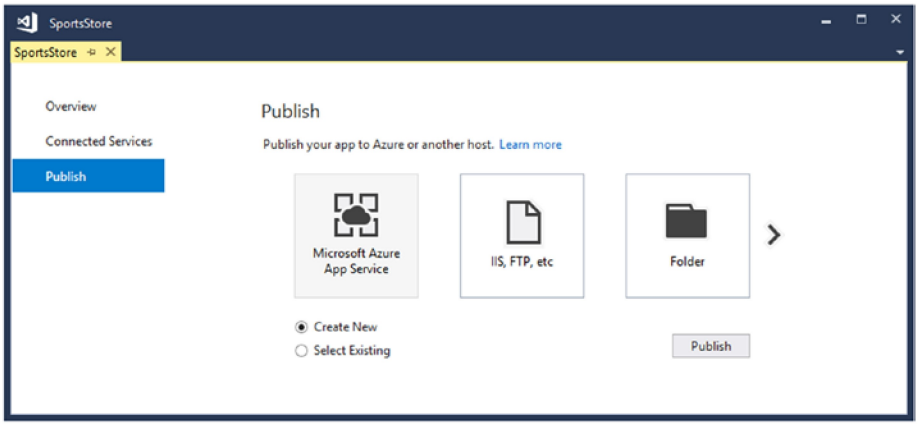

Figure 12-2. *Selecting a publishing method*

Select the Microsoft Azure App Service option and make sure that Create New is selected (the Select Existing option is used to update an existing deployed application). You will be prompted to provide details for the deployment. Start by clicking Add an Account and enter your Azure credentials.

Once you have entered your credentials, you can select a name for the deployed application and enter the details for the service, which will depend on the type of Azure account you have, the region you want to deploy to, and the deployment service you require, as shown in Figure 12-3.

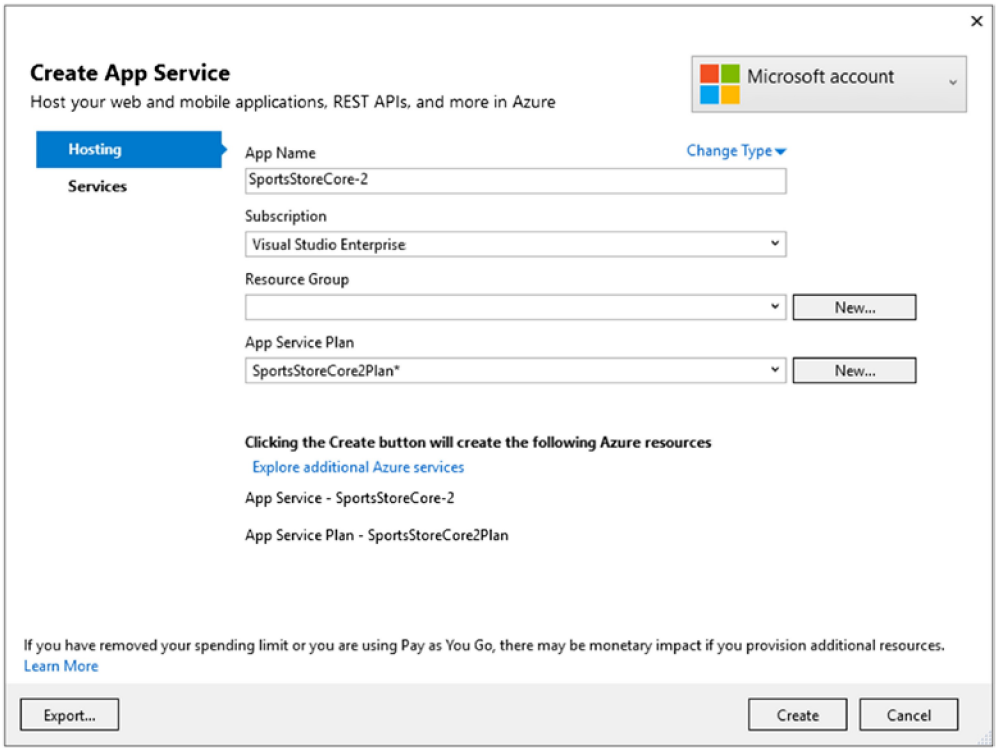

Figure 12-3. *Creating a new Azure app service*

Once you have configured the service, click the Create button. Once the service has been set up, you will be prompted with a summary of the publishing operation, which will send the application to the hosted service, as shown in Figure 12-4.

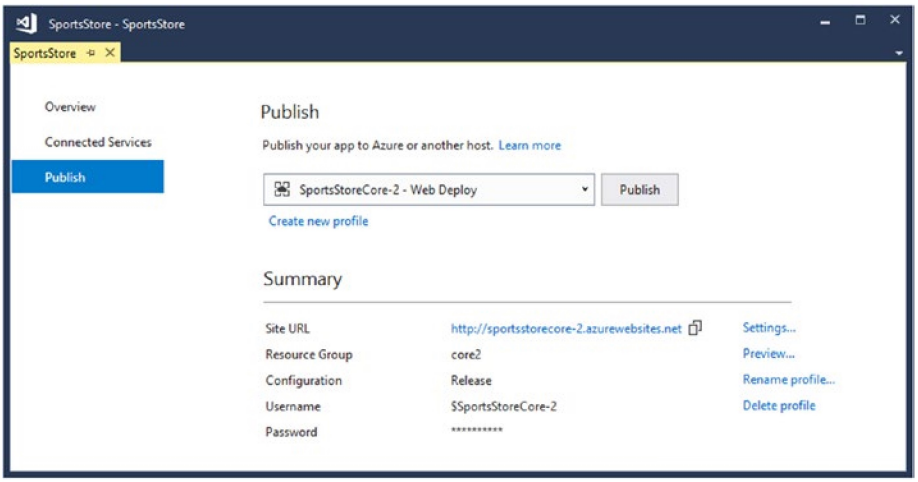

Figure 12-4. *The service publishing summary*

Click the Publish button to begin the deployment process. You can see details of the publishing progress by selecting Web Publish Activity from the Visual Studio View ➤ Other Windows menu. Be patient during this process because it can take a while to send all of the files in the project to the Azure service. Subsequent updates will be quicker because only modified files will be transferred.

Once deployment has completed, Visual Studio will open a new browser window for the deployed application. Since the product database is empty, you will see the layout shown in Figure 12-5.

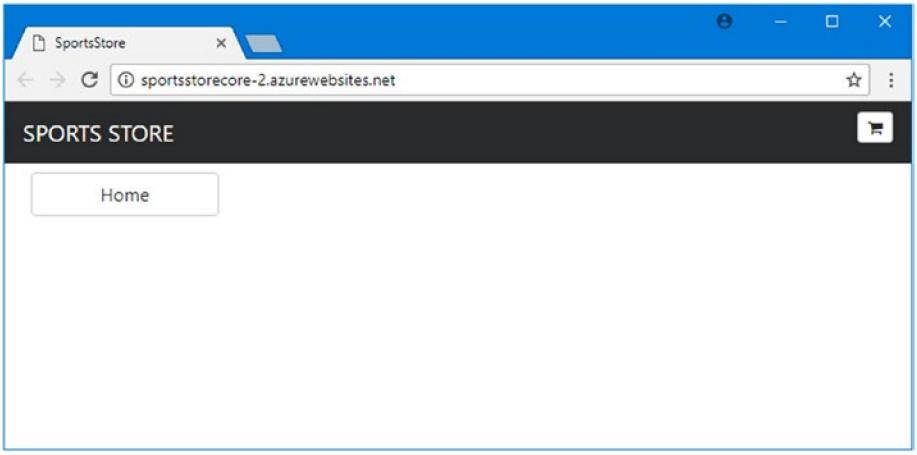

Figure 12-5. *The initial state of the deployed application*

Navigate to the `/Admin/Index` URL and authenticate with the username **Admin** and the password **Secret123$**. The Identity database will be seeded on-demand, allowing you to log into the administration part of the application, as shown in Figure 12-6.

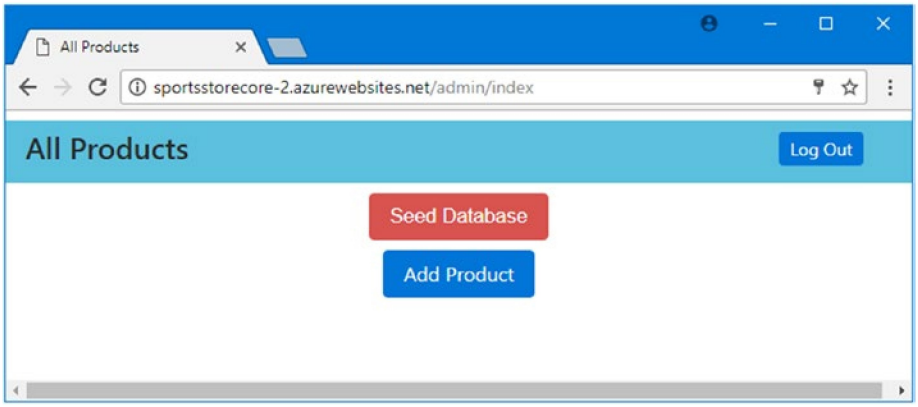

Figure 12-6. *The administration screen*

Click the Seed Database button to populate the product database, which will produce the result shown in Figure 12-7. You can then navigate back to the root URL for the application and use it as normal.

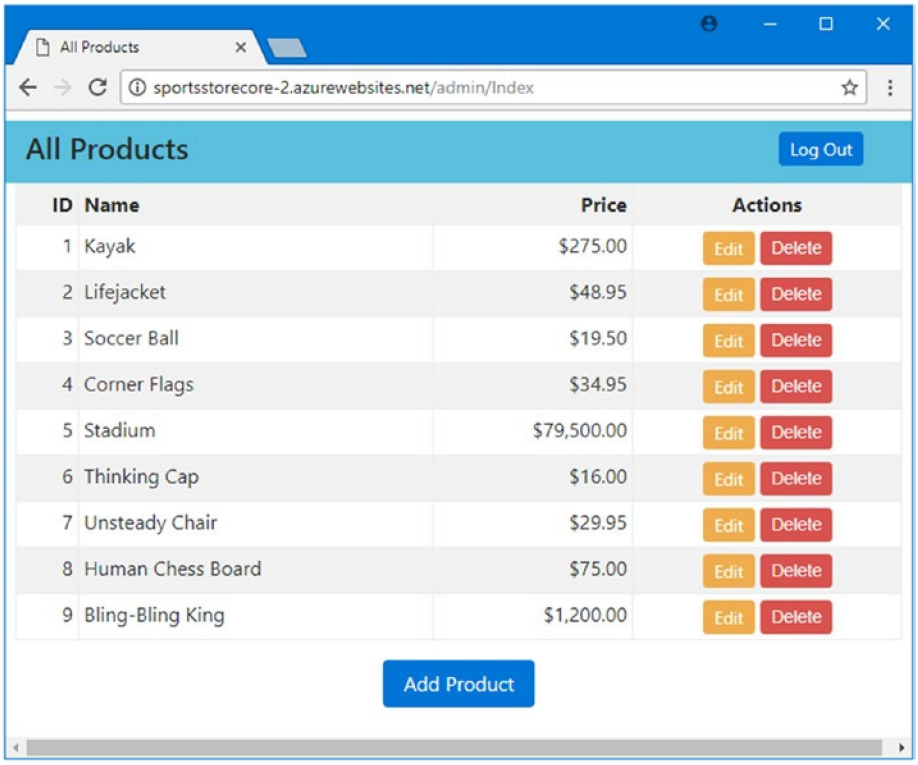

Figure 12-7. *The populated database*