

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.

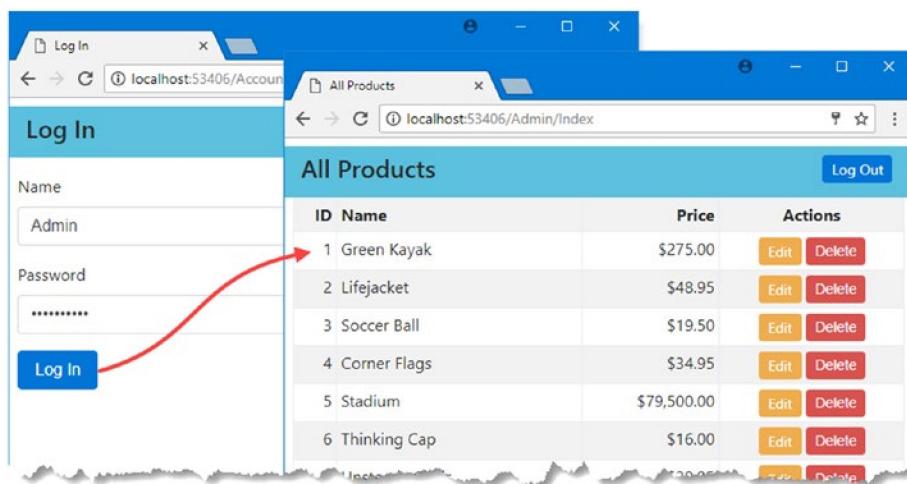


Figure 12-1. The administration authentication/authorization process

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};MultipleActiveResultsSets=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<ApplicationContext>(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 omitted 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.

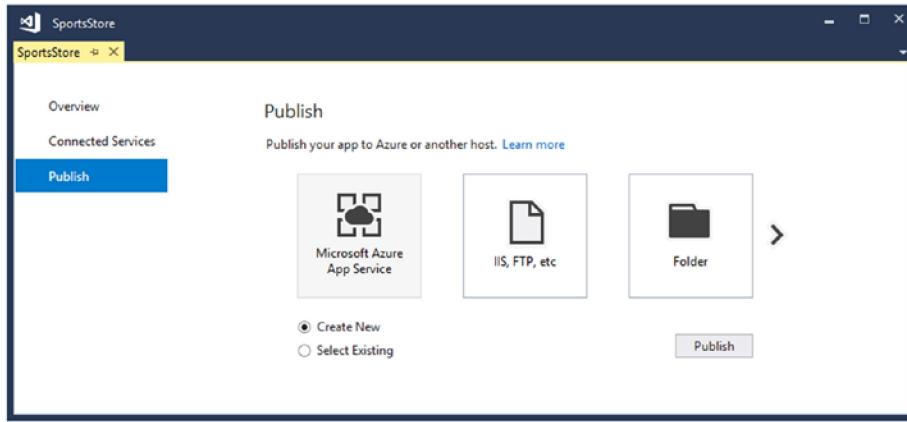


Figure 12-2. Selecting a publishing method

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.

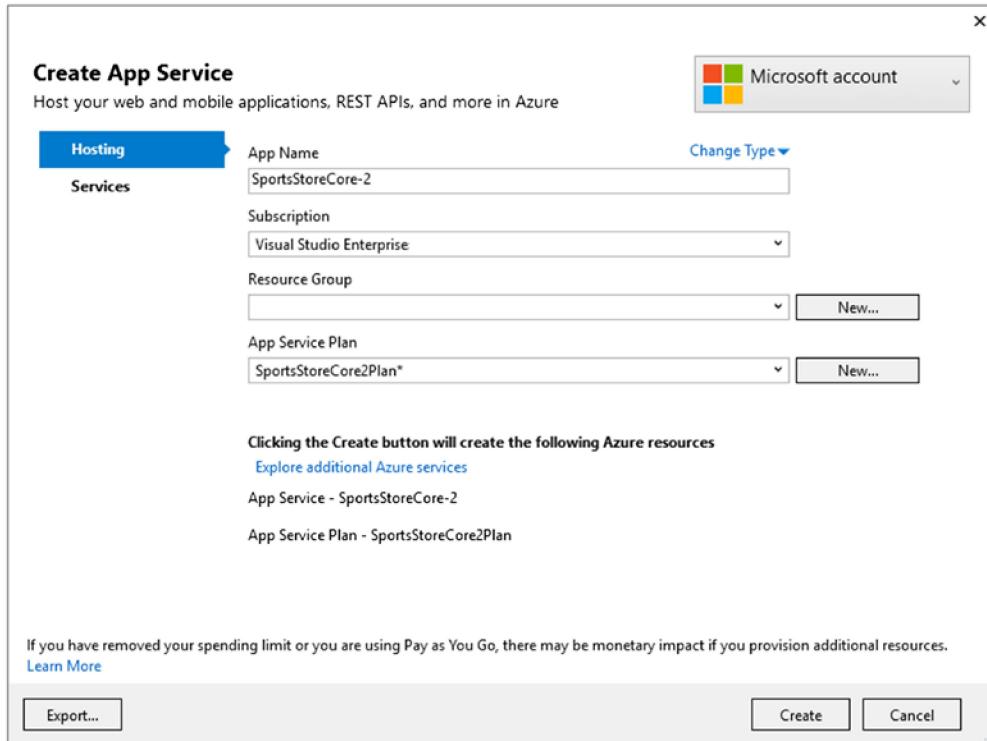


Figure 12-3. Creating a new Azure app service

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.

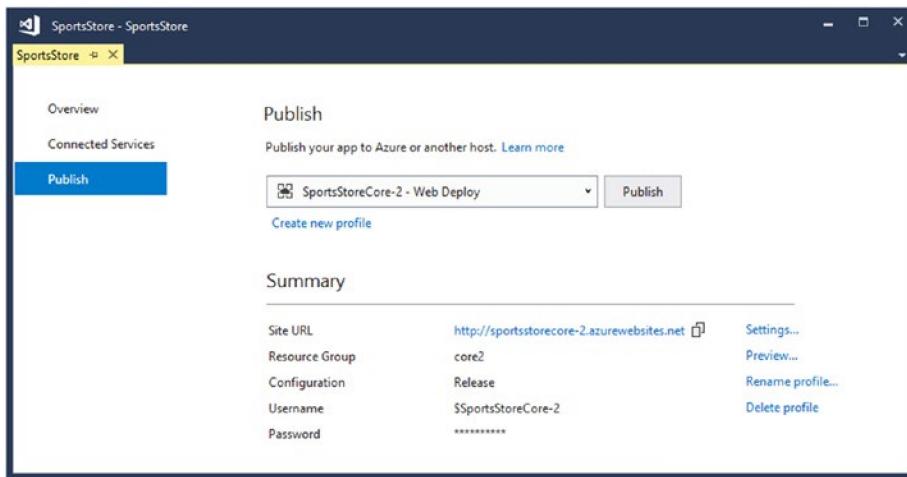


Figure 12-4. The service publishing summary

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.

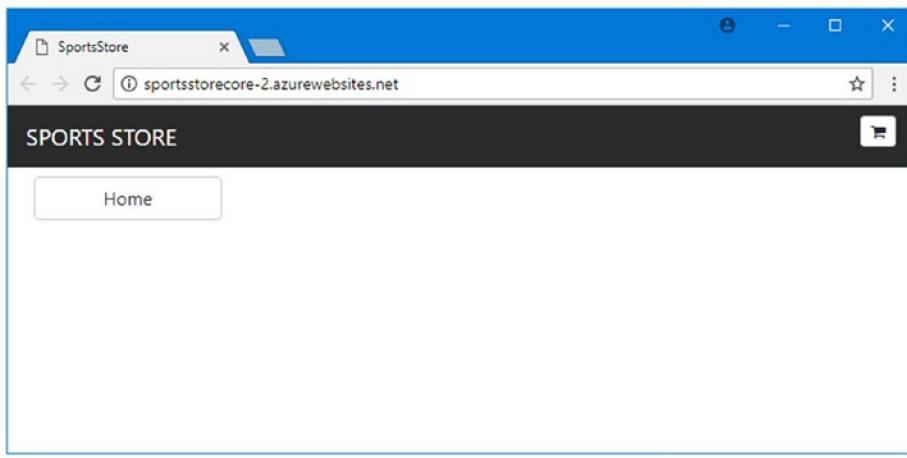


Figure 12-5. The initial state of the deployed application

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.

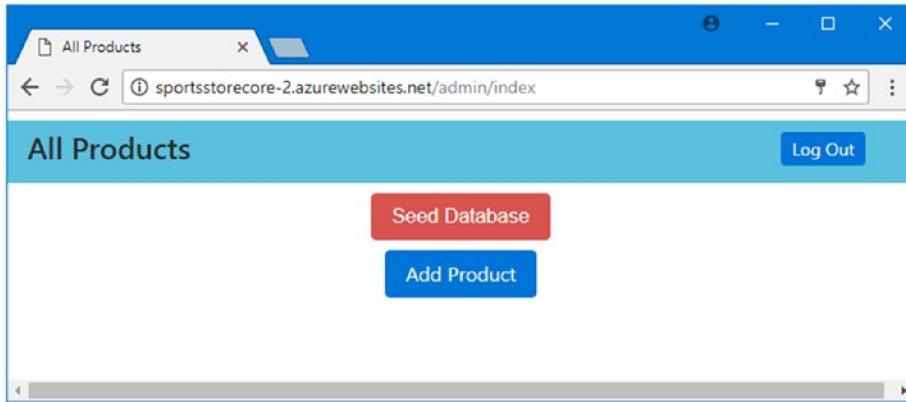


Figure 12-6. The administration screen

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.

ID	Name	Price	Actions	
1	Kayak	\$275.00	Edit	Delete
2	Lifejacket	\$48.95	Edit	Delete
3	Soccer Ball	\$19.50	Edit	Delete
4	Corner Flags	\$34.95	Edit	Delete
5	Stadium	\$79,500.00	Edit	Delete
6	Thinking Cap	\$16.00	Edit	Delete
7	Unsteady Chair	\$29.95	Edit	Delete
8	Human Chess Board	\$75.00	Edit	Delete
9	Bling-Bling King	\$1,200.00	Edit	Delete

[Add Product](#)

Figure 12-7. The populated database

Summary

In this and previous chapters, I demonstrated how 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 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. 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.