

Администрирование информационных систем

02.03.03 - Математическое обеспечение и администрирование информационных систем, направленность (профиль) - разработка и администрирование информационных систем

<http://vikchas.ru>

Лабораторные работы

Работа 8 Тема «Администрирование и защита ИС» английский

Часовских Виктор Петрович

доктор технических наук,
Профессор кафедры ШИиКМ
ФГБОУ ВО «Уральский государственный
экономический университет

Екатеринбург 2024



SportsStore: Administration

In this chapter, I continue to build the SportsStore application to give the site administrator a way of managing orders and products.

Managing Orders

In the previous chapter, I added support for receiving orders from customers and storing them in a database. In this chapter, I am going to create a simple administration tool that will let me view the orders that have been received and mark them as shipped.

Enhancing the Model

The first change I need to make is to enhance the model so that I can record which orders have been shipped. Listing 11-1 shows the addition of a new property to the `Order` class, which is defined in the `Order.cs` file in the `Models` folder.

Listing 11-1. Adding a Property in the `Order.cs` File in the `Models` Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Order {

        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [BindNever]
        public bool Shipped { get; set; }

        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]
        public string Line1 { get; set; }
    }
}
```

```

public string Line2 { get; set; }
public string Line3 { get; set; }

[Required(ErrorMessage = "Please enter a city name")]
public string City { get; set; }

[Required(ErrorMessage = "Please enter a state name")]
public string State { get; set; }

public string Zip { get; set; }

[Required(ErrorMessage = "Please enter a country name")]
public string Country { get; set; }

public bool GiftWrap { get; set; }
}
}

```

This iterative approach of extending and adapting the model to support different features is typical of MVC development. In an ideal world, you would be able to completely define the model classes at the start of the project and just build the application around them, but that happens only for the simplest of projects, and, in practice, iterative development is to be expected as the understanding of what is required develops and evolves.

Entity Framework Core migrations make this process easier because you don't have to manually keep the database schema synchronized to the model class by writing your own SQL commands. To update the database to reflect the addition of the Shipped property to the Order class, open a new command prompt or PowerShell window, navigate to the SportsStore project folder (which is the one that contains the Startup.cs file) and run the following command:

```
dotnet ef migrations add ShippedOrders
```

The migration will be applied automatically when the application is started and the SeedData class calls the Migrate method provided by Entity Framework Core.

Adding the Actions and View

The functionality required to display and update the set of orders in the database is relatively simple because it builds on the features and infrastructure that I created in previous chapters. In Listing 11-2, I have added two new action methods to the Order controller.

Listing 11-2. Adding Action Methods in the OrderController.cs File in the Controllers Folder

```

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

namespace SportsStore.Controllers {

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

```

```

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

public ActionResult List() =>
    View(repository.Orders.Where(o => !o.Shipped));

[HttpPost]
public ActionResult 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 ActionResult Checkout() => View(new Order());

[HttpPost]
public ActionResult 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 ActionResult Completed() {
    cart.Clear();
    return View();
}
}
}

```

The `List` method selects all the `Order` objects in the repository that have a `Shipped` value of `false` and passes them to the default view. This is the action method that I will use to display a list of the unshipped orders to the administrator.

The `MarkShipped` method will receive a POST request that specifies the ID of an order, which is used to locate the corresponding `Order` object from the repository so that the `Shipped` property can be set to `true` and saved.

To display the list of unshipped orders, I added a Razor view file called `List.cshtml` to the `Views/Order` folder and added the markup shown in Listing 11-3. A table element is used to display some of the details from each other, including details of which products have been purchased.

Listing 11-3. The Contents of the List.cshtml File in the Views/Order Folder

```
@model IEnumerable<Order>

@{
    ViewBag.Title = "Orders";
    Layout = "_AdminLayout";
}

@if (Model.Count() > 0) {

    <table class="table table-bordered table-striped">
        <tr><th>Name</th><th>Zip</th><th colspan="2">Details</th></tr>
        @foreach (Order o in Model) {
            <tr>
                <td>@o.Name</td><td>@o.Zip</td><th>Product</th><th>Quantity</th>
                <td>
                    <form asp-action="MarkShipped" method="post">
                        <input type="hidden" name="orderId" value="@o.OrderID" />
                        <button type="submit" class="btn btn-sm btn-danger">
                            Ship
                        </button>
                    </form>
                </td>
            </tr>
            @foreach (CartLine line in o.Lines) {
                <tr>
                    <td colspan="2"></td>
                    <td>@line.Product.Name</td><td>@line.Quantity</td>
                    <td></td>
                </tr>
            }
        }
    </table>
} else {
    <div class="text-center">No Unshipped Orders</div>
}
```

Each order is displayed with a Ship button that submits a form to the MarkShipped action method. I specified a different layout for the List view using the Layout property, which overrides the layout specified in the `_ViewStart.cshtml` file.

To add the layout, I used the MVC View Layout Page item template to create a file called `_AdminLayout.cshtml` in the Views/Shared folder, and I added the markup shown in Listing 11-4.

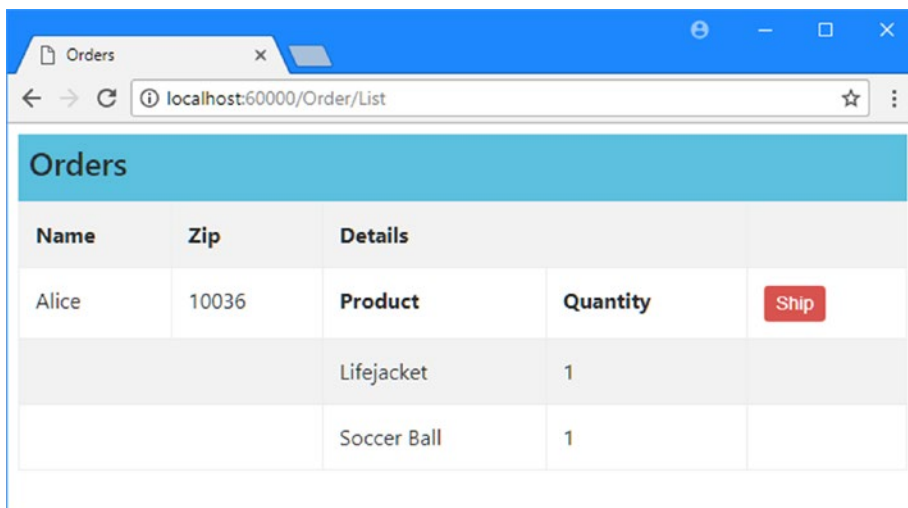
Listing 11-4. The Contents of the `_AdminLayout.cshtml` File in the Views/Shared Folder

```
<!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>
```

```
</head>
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @RenderBody()
</body>
</html>
```

To see and manage the orders in the application, start the application, select some products, and then check out. Then navigate to the `/Order/List` URL and you will see a summary of the order you created, as shown in Figure 11-1. Click the Ship button; the database will be updated, and the list of pending orders will be empty.

■ **Note** At the moment, there is nothing to stop customers from requesting the `/Order/List` URL and administering their own orders. I explain how to restrict access to action methods in Chapter 12.



Name	Zip	Details		
Alice	10036	Product	Quantity	Ship
		Lifejacket	1	
		Soccer Ball	1	

Figure 11-1. Managing orders

Adding Catalog Management

The convention for managing more complex collections of items is to present the user with two types of pages: a *list* page and an *edit* page, as shown in Figure 11-2.

List Screen	
Item	Actions
Kayak	Edit Delete
Lifejacket	Edit Delete
Soccer ball	Edit Delete
Add New Item	

Edit Item: Kayak	
Name:	<input type="text" value="Kayak"/>
Description:	<input type="text" value="A boat for one pe..."/>
Category:	<input type="text" value="Watersports"/>
Price (\$):	<input type="text" value="275.00"/>
Save Cancel	

Figure 11-2. Sketch of a CRUD UI for the product catalog

Together, these pages allow a user to create, read, update, and delete items in the collection. Collectively, these actions are known as *CRUD*. Developers need to implement CRUD so often that Visual Studio scaffolding includes scenarios for creating CRUD controllers with predefined action methods (I explained how to enable the scaffolding feature in Chapter 8). But like all the Visual Studio templates, I think it is better to learn how to use the features of the ASP.NET Core MVC directly.

Creating a CRUD Controller

I am going to start by creating a separate controller for managing the product catalog. I added a class file called `AdminController.cs` to the `Controllers` folder and added the code shown in Listing 11-5.

Listing 11-5. The Contents of the `AdminController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

        public IActionResult Index() => View(repository.Products);
    }
}
```

The controller constructor declares a dependency on the `IProductRepository` interface, which will be resolved when instances are created. The controller defines a single action method, `Index`, that calls the `View` method to select the default view for the action, passing the set of products in the database as the view model.

UNIT TEST: THE INDEX ACTION

The behavior that I care about for the `Index` method of the `AdminController` is that it correctly returns the `Product` objects that are in the repository. I can test this by creating a mock repository implementation and comparing the test data with the data returned by the action method. Here is the unit test, which I placed into a new unit test file called `AdminControllerTests.cs` in the `SportsStore.UnitTests` project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class AdminControllerTests {

        [Fact]
        public void Index_Contains_All_Products() {
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"},
                new Product {ProductID = 3, Name = "P3"},
            }.AsQueryable<Product>());

            // Arrange - create a controller
            AdminController target = new AdminController(mock.Object);

            // Action
            Product[] result
                = GetViewModel<IEnumerable<Product>>(target.Index()).ToArray();

            // Assert
            Assert.Equal(3, result.Length);
            Assert.Equal("P1", result[0].Name);
            Assert.Equal("P2", result[1].Name);
            Assert.Equal("P3", result[2].Name);
        }

        private T GetViewModel<T>(ActionResult result) where T : class {
            return (result as ViewResult)?.ViewData.Model as T;
        }
    }
}
```

I added a `GetViewModel` method to the test to unpack the result from the action method and get the view model data. I'll be adding more tests that use this method later in the chapter.

Implementing the List View

The next step is to add a view for the Index action method of the Admin controller. I created the Views/Admin folder and added a Razor file called Index.cshtml, the contents of which are shown in Listing 11-6.

Listing 11-6. The Contents of the Index.cshtml File in the Views/Admin Folder

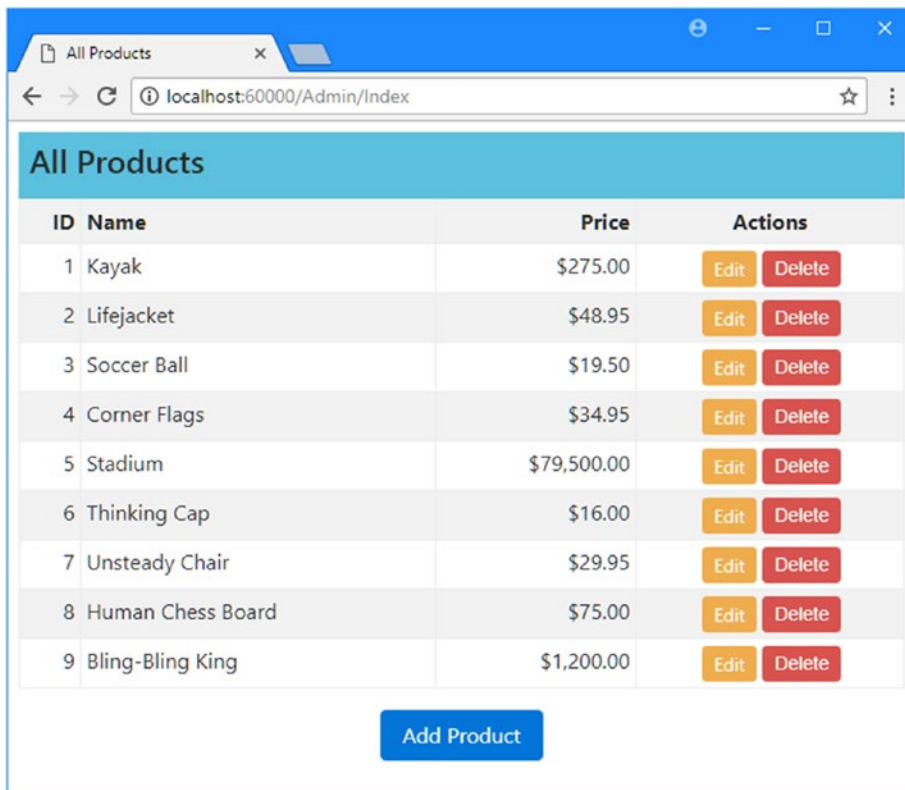
```
@model IEnumerable<Product>

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

<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>
```

This view contains a table that has a row for each product with cells that contain the name of the product, the price, and buttons that will allow the product to be edited or deleted by sending requests to Edit and Delete actions. In addition to the table, there is an Add Product button that targets the Create action. I'll add the Edit, Delete, and Create actions in the sections that follow, but you can see how the products are displayed by starting the application and requesting the /Admin/Index URL, as shown in Figure 11-3.

■ **Tip** The Edit button is inside the form element in Listing 11-6 so that the two buttons sit next to each other, working around the spacing that Bootstrap applies. The Edit button will send an HTTP GET request to the server to get the current details of a product; this doesn't require a `form` element. However, since the Delete button will make a change to the application state, I need to use an HTTP POST request—and that does require the `form` element.



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 11-3. *Displaying the list of products*

Editing Products

To provide create and update features, I will add a product-editing page like the one shown in Figure 11-2. These are the two parts of this job:

- Display a page that will allow the administrator to change values for the properties of a product
- Add an action method that can process those changes when they are submitted

Creating the Edit Action Method

Listing 11-7 shows the Edit action method I added to the Admin controller, which will receive the HTTP request sent by the browser when the user clicks an Edit button.

Listing 11-7. Adding an Edit Action Method in the AdminController.cs File in the Controllers Folder

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

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

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

This simple method finds the product with the ID that corresponds to the productId parameter and passes it as a view model object to the View method.

UNIT TEST: THE EDIT ACTION METHOD

I want to test for two behaviors in the Edit action method. The first is that I get the product I ask for when I provide a valid ID value to make sure that I am editing the product I expected. The second behavior to test is that I do not get any product at all when I request an ID value that is not in the repository. Here are the test methods I added to the AdminControllerTests.cs class file:

```
...
[Fact]
public void Can_Edit_Product() {
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
    }).AsQueryable<Product>());
```

```

// Arrange - create the controller
AdminController target = new AdminController(mock.Object);

// Act
Product p1 = GetViewModel<Product>(target.Edit(1));
Product p2 = GetViewModel<Product>(target.Edit(2));
Product p3 = GetViewModel<Product>(target.Edit(3));

// Assert
Assert.Equal(1, p1.ProductID);
Assert.Equal(2, p2.ProductID);
Assert.Equal(3, p3.ProductID);
}

[Fact]
public void Cannot_Edit_Nonexistent_Product() {
// Arrange - create the mock repository
Mock<IProductRepository> mock = new Mock<IProductRepository>();
mock.Setup(m => m.Products).Returns(new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
}.AsQueryable<Product>());

// Arrange - create the controller
AdminController target = new AdminController(mock.Object);

// Act
Product result = GetViewModel<Product>(target.Edit(4));

// Assert
Assert.Null(result);
}
...

```

Creating the Edit View

Now that I have an action method, I can create a view for it to display. I added a Razor view file called `Edit.cshtml` to the `Views/Admin` folder and added the markup shown in Listing 11-8.

Listing 11-8. The Contents of the `Edit.cshtml` File in the `Views/Admin` Folder

```

@model Product
@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />

```

```
<div class="form-group">
  <label asp-for="Name"></label>
  <input asp-for="Name" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Description"></label>
  <textarea asp-for="Description" class="form-control"></textarea>
</div>
<div class="form-group">
  <label asp-for="Category"></label>
  <input asp-for="Category" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Price"></label>
  <input asp-for="Price" class="form-control" />
</div>
<div class="text-center">
  <button class="btn btn-primary" type="submit">Save</button>
  <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</div>
</form>
```

The view contains an HTML form that uses tag helpers to generate much of the content, including setting the target for the form and a elements, setting the content of the label elements, and producing the name, id, and value attributes for the input and textarea elements.

You can see the HTML produced by the view by starting the application, navigating to the /Admin/Index URL, and clicking the Edit button for one of the products, as shown in Figure 11-4.

■ **Tip** I have used a hidden input element for the `ProductID` property for simplicity. The value of the `ProductID` is generated by the database as a primary key when a new object is stored by Entity Framework Core, and safely changing it can be a complex process. For most applications, the simplest approach is to prevent the user from changing the value.

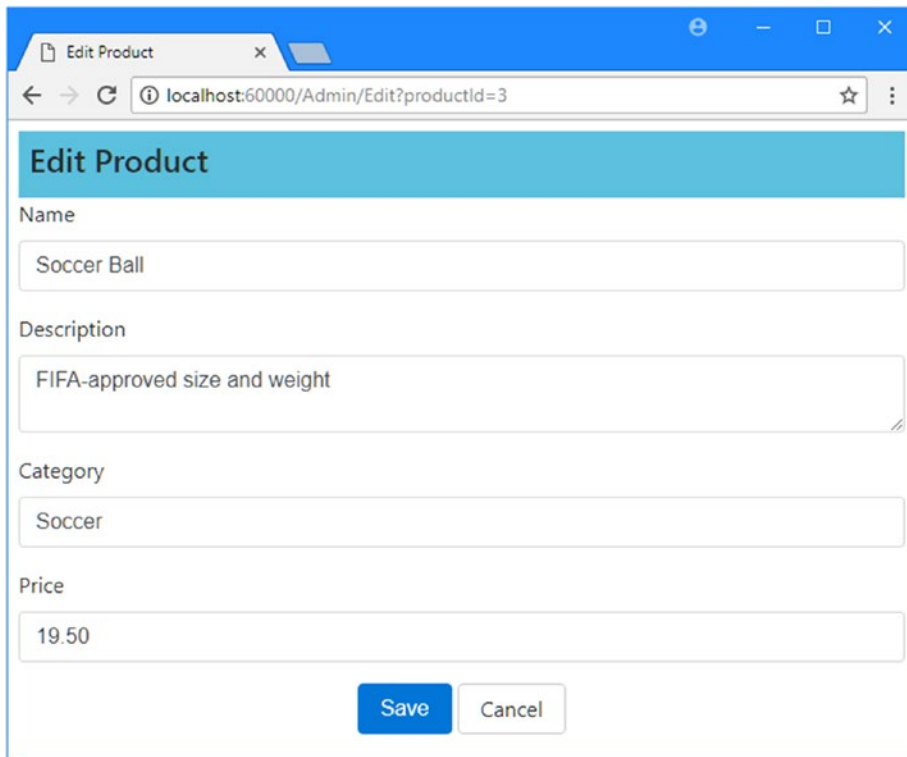


Figure 11-4. *Displaying product values for editing*

Updating the Product Repository

Before I can process edits, I need to enhance the product repository so that it is able to save changes. First, I added a new method to the `IProductRepository` interface, as shown in Listing 11-9.

Listing 11-9. Adding a Method to the `IProductRepository.cs` File in the Models Folder using `System.Linq`;

```
namespace SportsStore.Models {  
    public interface IProductRepository {  
        IQueryable<Product> Products { get; }  
        void SaveProduct(Product product);  
    }  
}
```

I can then add the new method to the Entity Framework Core implementation of the repository, which is defined in the `EFProductRepository.cs` file, as shown in Listing 11-10.

Listing 11-10. Implementing the New Method in the `EFProductRepository.cs` File in the Models Folder

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

namespace SportsStore.Models {

    public class EFProductRepository : IProductRepository {
        private ApplicationDbContext context;

        public EFProductRepository(ApplicationDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.Products;

        public void SaveProduct(Product product) {
            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                Product dbEntry = context.Products
                    .FirstOrDefault(p => p.ProductID == product.ProductID);
                if (dbEntry != null) {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Description;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;
                }
            }
            context.SaveChanges();
        }
    }
}
```

The implementation of the `SaveChanges` method adds a product to the repository if the `ProductID` is 0; otherwise, it applies any changes to the existing entry in the database.

I do not want to go into details of Entity Framework Core because, as I explained earlier, it is a topic in itself and not part of ASP.NET Core MVC. But there is something in the `SaveProduct` method that has a bearing on the design of the MVC application.

I know I need to perform an update when I receive a `Product` parameter that has a `ProductID` that is not zero. I do this by getting a `Product` object from the repository with the same `ProductID` and updating each of the properties so they match the parameter object.

I can do this because Entity Framework Core keeps track of the objects that it creates from the database. The object passed to the `SaveChanges` method is created by the MVC model binding feature, which means that Entity Framework Core does not know anything about the new `Product` object and will not apply an update to the database when it is modified. There are lots of ways of resolving this issue, and I have taken the simplest one, which is to locate the corresponding object that Entity Framework Core *does* know about and update it explicitly.

The addition of a new method in the `IProductRepository` interface has broken the fake repository class—`FakeProductRepository`—that I created in Chapter 8. I used the fake repository to kick-start the development process and demonstrate how services can be used to seamlessly replace interface implementations without needing to modify the components that rely on them. I don't need the fake repository any further, and in Listing 11-11, you can see that I have removed the interface from the class declaration so that I don't have to keep modifying the class as I add repository features.

Listing 11-11. Removing the Interface in the `FakeProductRepository.cs` File in the Models Folder

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

namespace SportsStore.Models {

    public class FakeProductRepository /* : IProductRepository */ {

        public IQueryable<Product> Products => new List<Product> {
            new Product { Name = "Football", Price = 25 },
            new Product { Name = "Surf board", Price = 179 },
            new Product { Name = "Running shoes", Price = 95 }
        }.AsQueryable<Product>();
    }
}
```

Handling Edit POST Requests

I am ready to implement an overload of the `Edit` action method in the `AdminController` that will handle POST requests when the administrator clicks the Save button. Listing 11-12 shows the new action method.

Listing 11-12. Defining an Action Method in the `AdminController.cs` File in the Controllers Folder

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

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

        public IActionResult 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);
        }
    }
}

```

I check that the model binding process has been able to validate the data submitted by the user by reading the value of the `ModelState.IsValid` property. If everything is OK, I save the changes to the repository and redirect the user to the `Index` action so they see the modified list of products. If there is a problem with the data, I render the default view again so that the user can make corrections.

After I have saved the changes in the repository, I store a message using the *temp data* feature, which is part of the ASP.NET Core session state feature. This is a key/value dictionary similar to the session data and view bag features I used previously. The key difference from session data is that temp data persists until it is read.

I cannot use `ViewBag` in this situation because `ViewBag` passes data between the controller and view and it cannot hold data for longer than the current HTTP request. When an edit succeeds, the browser is redirected to a new URL, so the `ViewBag` data is lost. I could use the session data feature, but then the message would be persistent until I explicitly removed it, which I would rather not have to do.

So, the temp data feature is the perfect fit. The data is restricted to a single user's session (so that users do not see each other's `TempData`) and will persist long enough for me to read it. I will read the data in the view rendered by the action method to which I have redirected the user, which I define in the next section.

UNIT TEST: EDIT SUBMISSIONS

For the POST-processing `Edit` action method, I need to make sure that valid updates to the `Product` object, which is received as the method argument, are passed to the product repository to be saved. I also want to check that invalid updates (where a model validation error exists) are not passed to the repository. Here are the test methods, which I added to the `AdminControllerTests.cs` file:

```

...
[Fact]
public void Can_Save_Valid_Changes() {
    // Arrange - create mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    // Arrange - create mock temp data
    Mock<ITempDataDictionary> tempData = new Mock<ITempDataDictionary>();
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object) {
        TempData = tempData.Object
    };
    // Arrange - create a product
    Product product = new Product { Name = "Test" };

    // Act - try to save the product
    IActionResult result = target.Edit(product);
}

```

```

        // Assert - check that the repository was called
        mock.Verify(m => m.SaveProduct(product));
        // Assert - check the result type is a redirection
        Assert.IsType<RedirectToActionResult>(result);
        Assert.Equal("Index", (result as RedirectToActionResult).ActionName);
    }

    [Fact]
    public void Cannot_Save_Invalid_Changes() {
        // Arrange - create mock repository
        Mock<IProductRepository> mock = new Mock<IProductRepository>();
        // Arrange - create the controller
        AdminController target = new AdminController(mock.Object);
        // Arrange - create a product
        Product product = new Product { Name = "Test" };
        // Arrange - add an error to the model state
        target.ModelState.AddModelError("error", "error");

        // Act - try to save the product
        IActionResult result = target.Edit(product);

        // Assert - check that the repository was not called
        mock.Verify(m => m.SaveProduct(It.IsAny<Product>()), Times.Never());
        // Assert - check the method result type
        Assert.IsType<ViewResult>(result);
    }
    ...

```

Displaying a Confirmation Message

I am going to deal with the message I stored using TempData in the `_AdminLayout.cshtml` layout file, as shown in Listing 11-13. By handling the message in the template, I can create messages in any view that uses the template without needing to create additional Razor expressions.

Listing 11-13. Handling the ViewBag Message 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>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
    @if (TempData["message"] != null) {
        <div class="alert alert-success">@TempData["message"]</div>
    }
    @RenderBody()
</body>
</html>

```

Tip The benefit of dealing with the message in the template like this is that users will see it displayed on whatever page is rendered after they have saved a change. At the moment, I return them to the list of products, but I could change the workflow to render some other view, and the users will still see the message (as long as the next view also uses the same layout).

I now have all the pieces in place to edit products. To see how it all works, start the application, navigate to the /Admin/Index URL, click the Edit button, and make a change. Click the Save button. You will be redirected to the /Admin/Index URL, and the TempData message will be displayed, as shown in Figure 11-5. The message will disappear if you reload the product list screen because TempData is deleted when it is read. That is convenient since I do not want old messages hanging around.

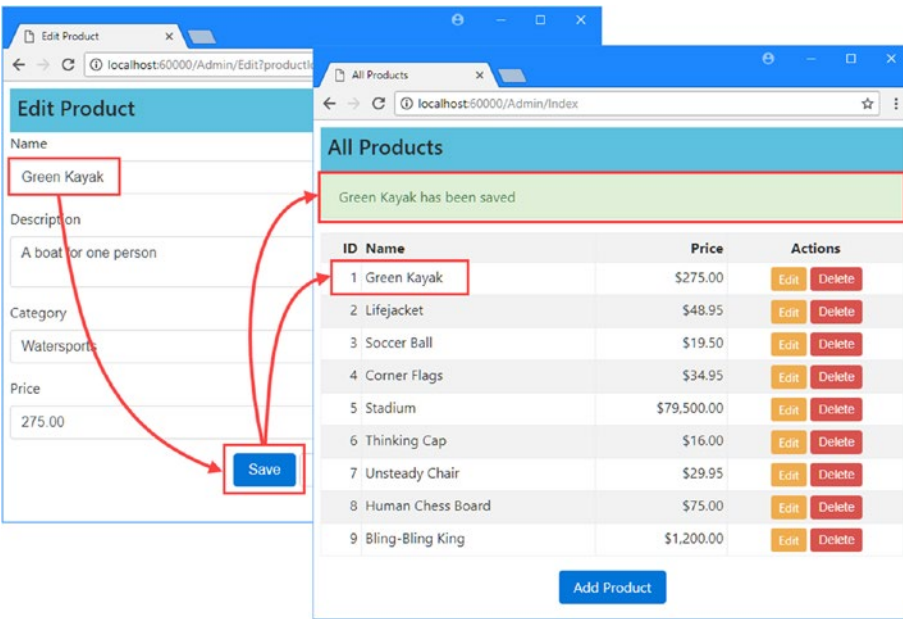


Figure 11-5. Editing a product and seeing the TempData message

Adding Model Validation

I have reached the point where I need to add validation rules to the model classes. At the moment, the administrator could enter negative prices or blank descriptions, and SportsStore would happily store that data in the database. Whether or not the bad data would be successfully persisted would depend on whether it conformed to the constraints in the SQL tables created by Entity Framework Core, and that is not enough protection for most applications. To guard against bad data values, I decorated the properties of the Product class with attributes, as shown in Listing 11-14, just as I did for the Order class in Chapter 10.

Listing 11-14. Applying Validation Attributes in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Product {
        public int ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue,
            ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }
    }
}
```

In Chapter 10, I used a tag helper to display a summary of validation errors at the top of the form. For this example, I am going to use a similar approach, but I am going to display error messages next to individual form elements in the Edit view, as shown in Listing 11-15.

Listing 11-15. Adding Validation Error Elements in the Edit.cshtml File in the Views/Admin Folder

```
@model Product
@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />
    <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="Description"></label>
        <div><span asp-validation-for="Description" class="text-danger"></span></div>
        <textarea asp-for="Description" class="form-control"></textarea>
    </div>
    <div class="form-group">
        <label asp-for="Category"></label>
```

```

    <div><span asp-validation-for="Category" class="text-danger"></span></div>
    <input asp-for="Category" class="form-control" />
</div>
<div class="form-group">
    <label asp-for="Price"></label>
    <div><span asp-validation-for="Price" class="text-danger"></span></div>
    <input asp-for="Price" class="form-control" />
</div>
<div class="text-center">
    <button class="btn btn-primary" type="submit">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
</div>
</form>

```

When applied to a span element, the `asp-validation-for` attribute applies a tag helper that will add a validation error message for the specified property if there are any validation problems.

The tag helpers will insert an error message into the span element and add the element to the `input-validation-error` class, which makes it easy to apply CSS styles to error message elements, as shown in Listing 11-16.

Listing 11-16. Adding CSS to the `_AdminLayout.cshtml` File in the Views/Shared Folder

```

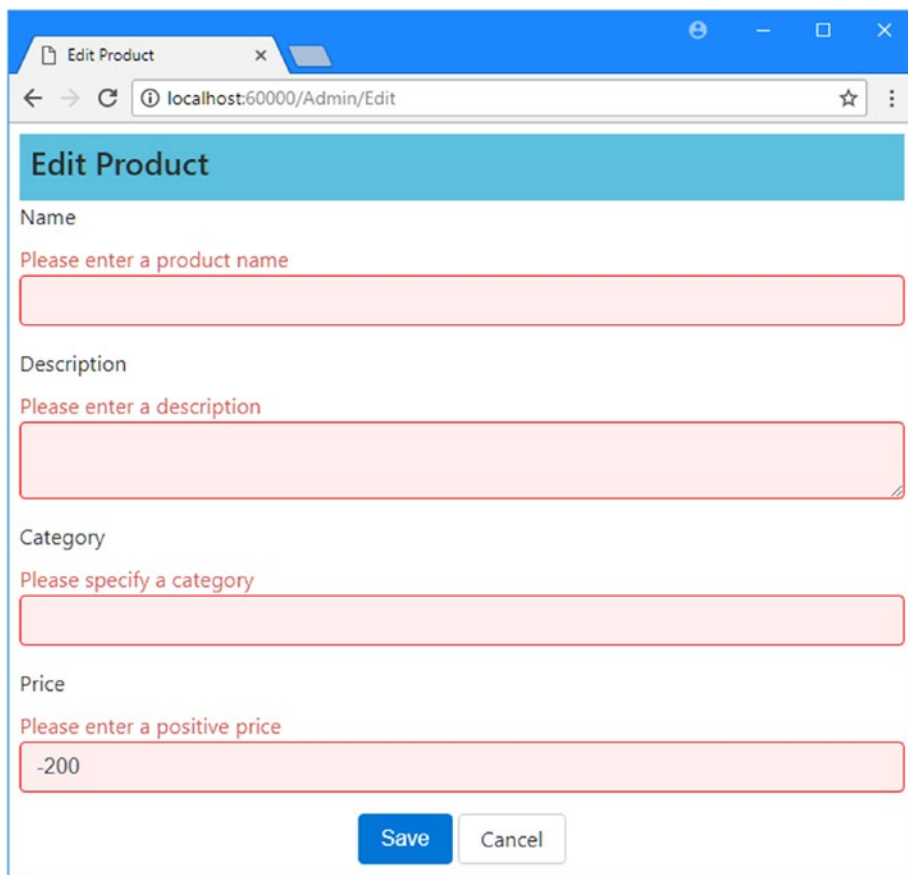
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
    <style>
        .input-validation-error { border-color: red; background-color: #fee ; }
    </style>
</head>
<body class="m-1 p-1">
    <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
    @if (TempData["message"] != null) {
        <div class="alert alert-success mt-1">@TempData["message"]</div>
    }
    @RenderBody()
</body>
</html>

```

The CSS style I defined selects elements that are members of the `input-validation-error` class and applies a red border and background color.

■ **Tip** Explicitly setting styles when using a CSS library like Bootstrap can cause inconsistencies when content themes are applied. In Chapter 27, I show an alternative approach that uses JavaScript code to apply Bootstrap classes to elements with validation errors, which keeps everything consistent but is also more complex.

You can apply the validation message tag helpers anywhere in the view, but it is conventional (and sensible) to put it somewhere near the problem element to give the user some context. Figure 11-6 shows the validation messages and cues that are displayed, which you can see by running the application, editing a product, and submitting invalid data.



The screenshot shows a web browser window titled "Edit Product" with the URL "localhost:60000/Admin/Edit". The page has a blue header with the title "Edit Product". Below the header, there are four form fields, each with a red border and a red validation message above it:

- Name:** "Please enter a product name". The input field is empty.
- Description:** "Please enter a description". The input field is empty.
- Category:** "Please specify a category". The input field is empty.
- Price:** "Please enter a positive price". The input field contains the value "-200".

At the bottom of the form, there are two buttons: a blue "Save" button and a white "Cancel" button with a grey border.

Figure 11-6. Data validation when editing products

Enabling Client-Side Validation

Currently, data validation is applied only when the administration user submits edits to the server, but most users expect immediate feedback if there are problems with the data they have entered. This is why developers often want to perform *client-side validation*, where the data is checked in the browser using JavaScript. MVC applications can perform client-side validation based on the data annotations I applied to the domain model class.

The first step is to add the JavaScript libraries that provide the client-side feature to the application, which is done in the `bower.json` file, as shown in Listing 11-17.

Listing 11-17. Adding JavaScript Packages in the bower.json File

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Client-side validation is built on top of the popular jQuery library, which simplifies working with the browser's DOM API. The next step is to add the JavaScript files to the layout so they are loaded when the SportsStore administration features are used, as shown in Listing 11-18.

Listing 11-18. Adding the Validation Libraries to the _AdminLayout.cshtml File in the Views/Shared Folder

```
<!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"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Enabling client-side validation doesn't cause any visual change, but the constraints specified by the attributes applied to the C# model class are enforced at the browser, preventing the user from submitting the form with bad data and providing immediate feedback when there is a problem. See Chapter 27 for more details.

Creating New Products

Next, I will implement the Create action method, which is the one specified by the Add Product link in the main product list page. This will allow the administrator to add new items to the product catalog. Adding the ability to create new products will require one small addition to the application. This is a great example of the power and flexibility of a well-structured MVC application. First, add the Create method, shown in Listing 11-19, to the Admin controller.

Listing 11-19. Adding the Create Action to the AdminController.cs File in the Controllers Folder

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

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

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

The Create method does not render its default view. Instead, it specifies that the Edit view should be used. It is perfectly acceptable for one action method to use a view that is usually associated with another view. In this case, I provide a new Product object as the view model so that the Edit view is populated with empty fields.

Note I have not added a unit test for this action method. Doing so would only be testing the ASP.NET Core MVC ability to process the result from the action method result, which is something you can take for granted. (Tests are not usually written for framework features unless you suspect there is a defect.)

That is the only change that is required because the `Edit` action method is already set up to receive `Product` objects from the model binding system and store them in the database. You can test this functionality by starting the application, navigating to `/Admin/Index`, clicking the `Add Product` button, and populating and submitting the form. The details you specify in the form will be used to create a new product in the database, which will then appear in the list, as shown in Figure 11-7.

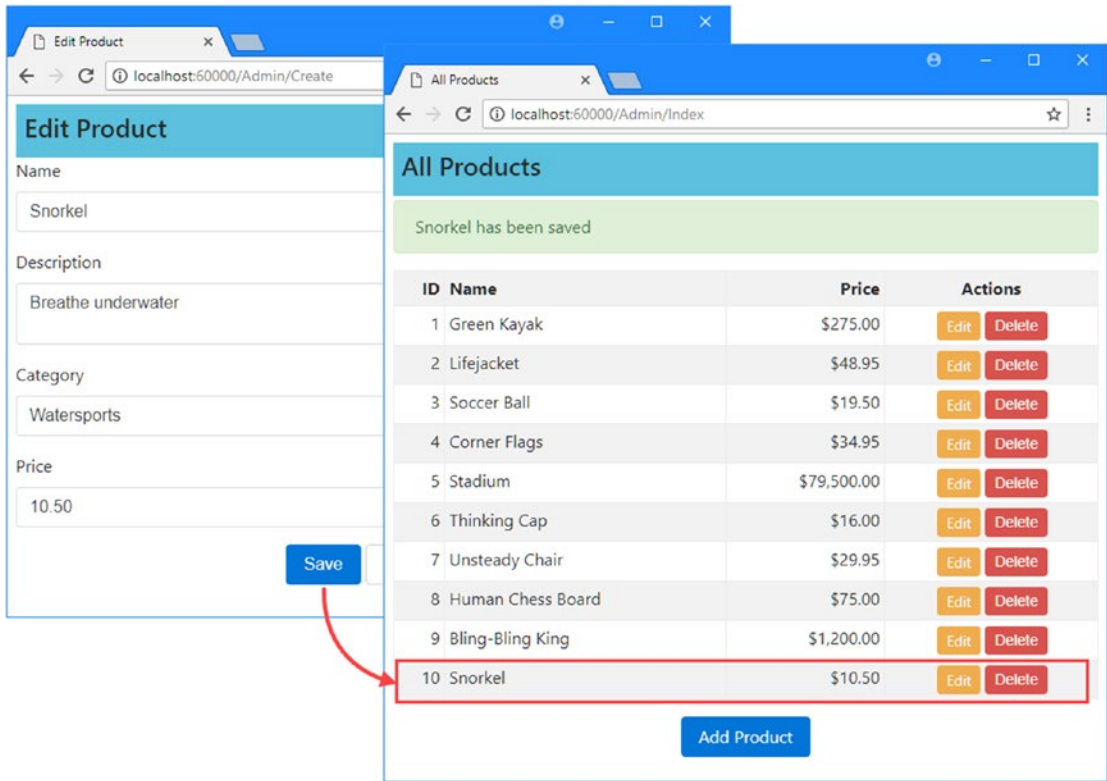


Figure 11-7. Adding a new product to the catalog

Deleting Products

Adding support for deleting items is also simple. The first step is to add a new method to the `IProductRepository` interface, as shown in Listing 11-20.

Listing 11-20. Adding a Method to Delete Products to the IProductRepository.cs File in the Models Folder
using System.Linq;

```
namespace SportsStore.Models {  
  
    public interface IProductRepository {  
  
        IQueryable<Product> Products { get; }  
  
        void SaveProduct(Product product);  
  
        Product DeleteProduct(int productID);  
    }  
}
```

Next, I implement this method in the Entity Framework Core repository class, EFProductRepository, as shown in Listing 11-21.

Listing 11-21. Implementing Deletion Support in the EFProductRepository.cs File in the Models Folder

```
using System.Collections.Generic;  
using System.Linq;  
  
namespace SportsStore.Models {  
  
    public class EFProductRepository : IProductRepository {  
        private ApplicationDbContext context;  
  
        public EFProductRepository(ApplicationDbContext ctx) {  
            context = ctx;  
        }  
  
        public IQueryable<Product> Products => context.Products;  
  
        public void SaveProduct(Product product) {  
            if (product.ProductID == 0) {  
                context.Products.Add(product);  
            } else {  
                Product dbEntry = context.Products  
                    .FirstOrDefault(p => p.ProductID == product.ProductID);  
                if (dbEntry != null) {  
                    dbEntry.Name = product.Name;  
                    dbEntry.Description = product.Description;  
                    dbEntry.Price = product.Price;  
                    dbEntry.Category = product.Category;  
                }  
            }  
            context.SaveChanges();  
        }  
    }  
}
```

```

    public Product DeleteProduct(int productID) {
        Product dbEntry = context.Products
            .FirstOrDefault(p => p.ProductID == productID);
        if (dbEntry != null) {
            context.Products.Remove(dbEntry);
            context.SaveChanges();
        }
        return dbEntry;
    }
}
}
}

```

The final step is to implement a Delete action method in the Admin controller. This action method should support only POST requests because deleting objects is not an idempotent operation. As I explain in Chapter 16, browsers and caches are free to make GET requests without the user's explicit consent, so I must be careful to avoid making changes as a consequence of GET requests. Listing 11-22 shows the new action method.

Listing 11-22. Adding the Delete Action Method in the AdminController.cs File in the Controllers Folder

```

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

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

        public IActionResult 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 IActionResult 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");
}
}
}

```

UNIT TEST: DELETING PRODUCTS

I want to test the basic behavior of the Delete action method, which is that when a valid ProductID is passed as a parameter, the action method calls the DeleteProduct method of the repository and passes the correct ProductID value to be deleted. Here is the test that I added to the AdminControllerTests.cs file:

```

...
[Fact]
public void Can_Delete_Valid_Products() {
    // Arrange - create a Product
    Product prod = new Product { ProductID = 2, Name = "Test" };

    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        prod,
        new Product {ProductID = 3, Name = "P3"},
    }.AsQueryable<Product>());

    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);

    // Act - delete the product
    target.Delete(prod.ProductID);

    // Assert - ensure that the repository delete method was
    // called with the correct Product
    mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
...

```

You can see the delete feature by starting the application, navigating to /Admin/Index, and clicking one of the Delete buttons in the product list page, as shown in Figure 11-8. As shown in the figure, I have taken advantage of the TempData variable to display a message when a product is deleted from the catalog.

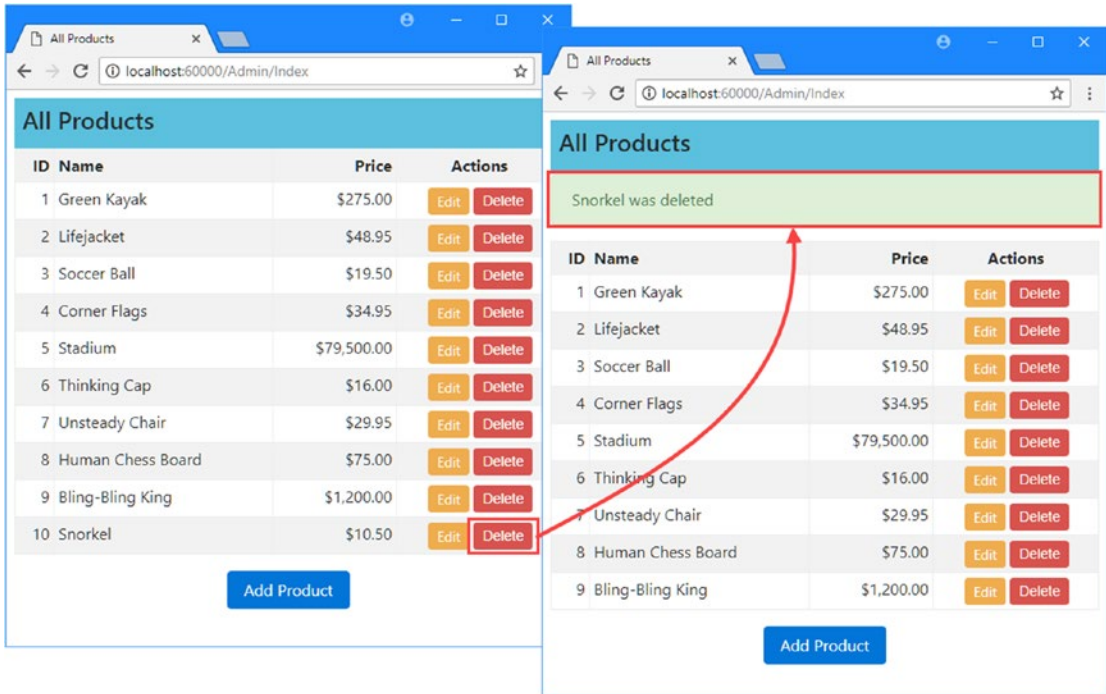


Figure 11-8. Deleting a product from the catalog

Note You will find that you get an error if you delete a product for which you have previously created an order. When an Order object is stored in the database, it is transformed into an entry in a database table that contains a reference to the Product object with which it is associated, known as a *foreign key relationship*. This means that, by default, the database won't allow a Product object to be deleted if an Order has been created for that Product because doing so would create an inconsistency in the database. There are a number of ways to approach this issue, including automatically deleting Order objects when the Product they relate to is deleted or changing the relationship between Product and Order objects. See the Entity Framework Core documentation for details.

Summary

In this chapter, I introduced the administration capability and showed you how to implement CRUD operations that allow the administrator to create, read, update, and delete products from the repository and mark orders as shipped. In the next chapter, I show you how to secure the administration functions so that they are not available to all users, and I deploy the SportsStore application into production.



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.

■ **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 `SportStoreIdentity` 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 IActionResult 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 IActionResult 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 IActionResult 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 IActionResult Index() => View(repository.Products);

        public IActionResult 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 IActionResult 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 IActionResult 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>
    <div>
        @if (TempData["message"] != null) {
            <div class="alert alert-success mt-1">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>

```


Testing the Security Policy

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

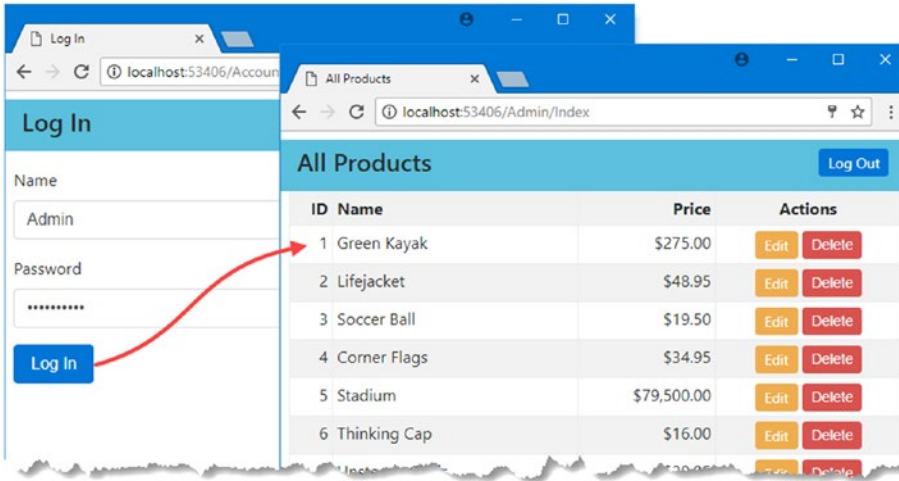


Figure 12-1. The administration authentication/authorization process

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};MultipleActiveResultSets=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 IActionResult 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 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 IActionResult 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.

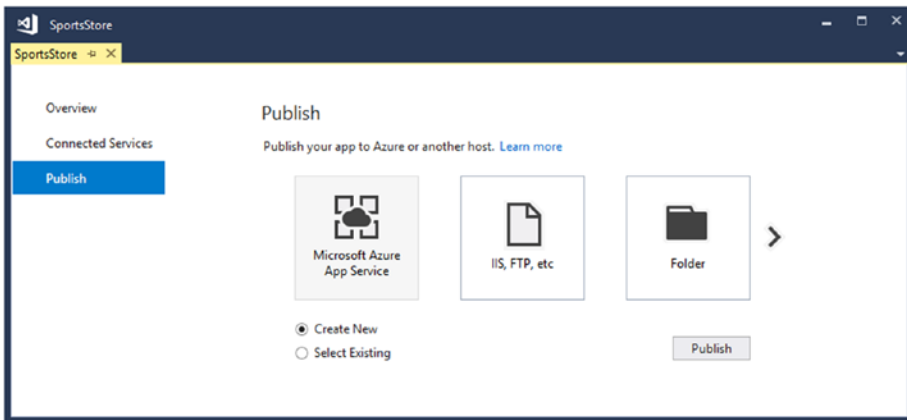


Figure 12-2. Selecting a publishing method

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.

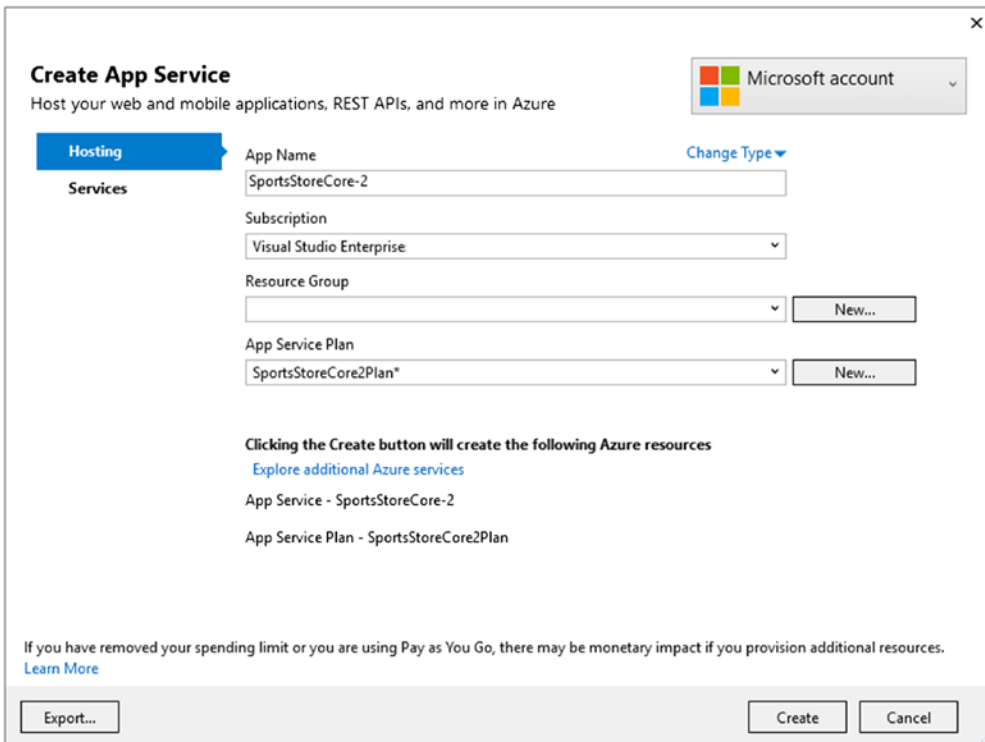


Figure 12-3. Creating a new Azure app service

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.

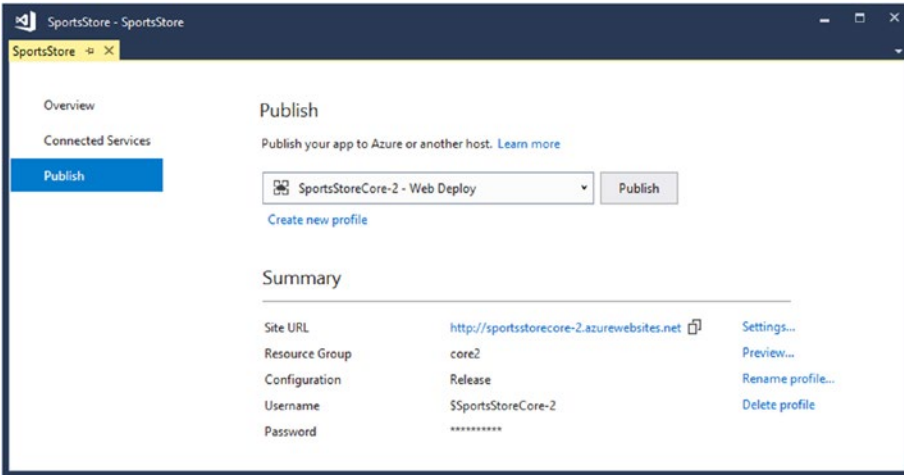


Figure 12-4. The service publishing summary

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.

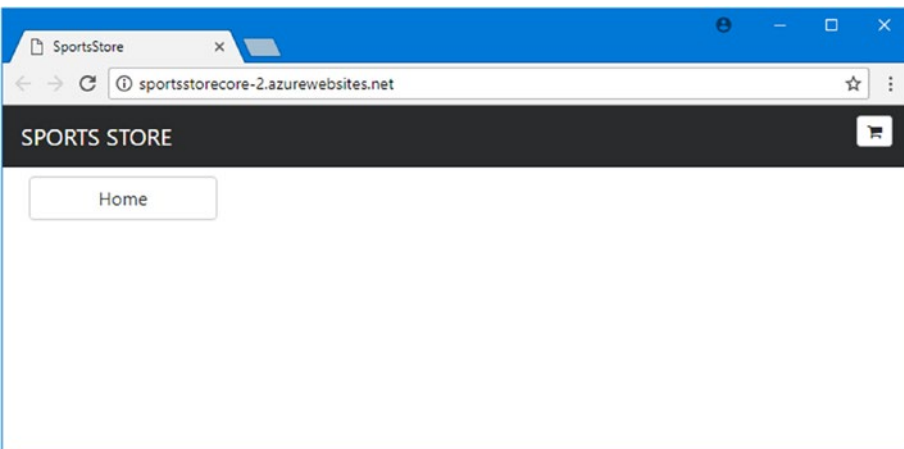


Figure 12-5. The initial state of the deployed application

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.

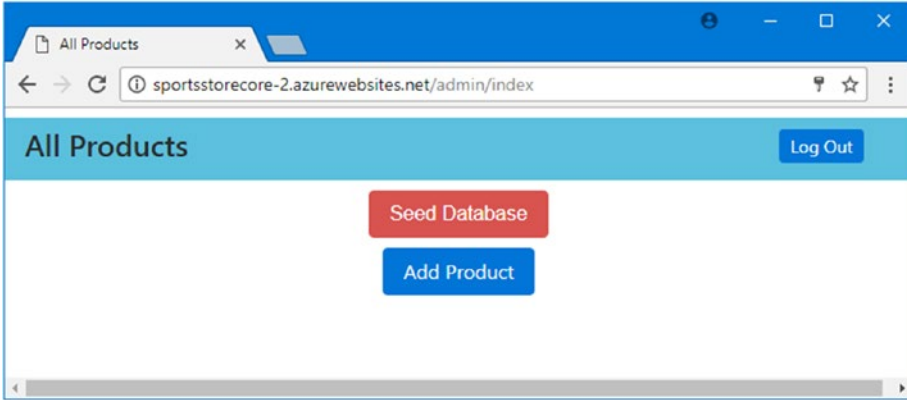


Figure 12-6. The administration screen

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.

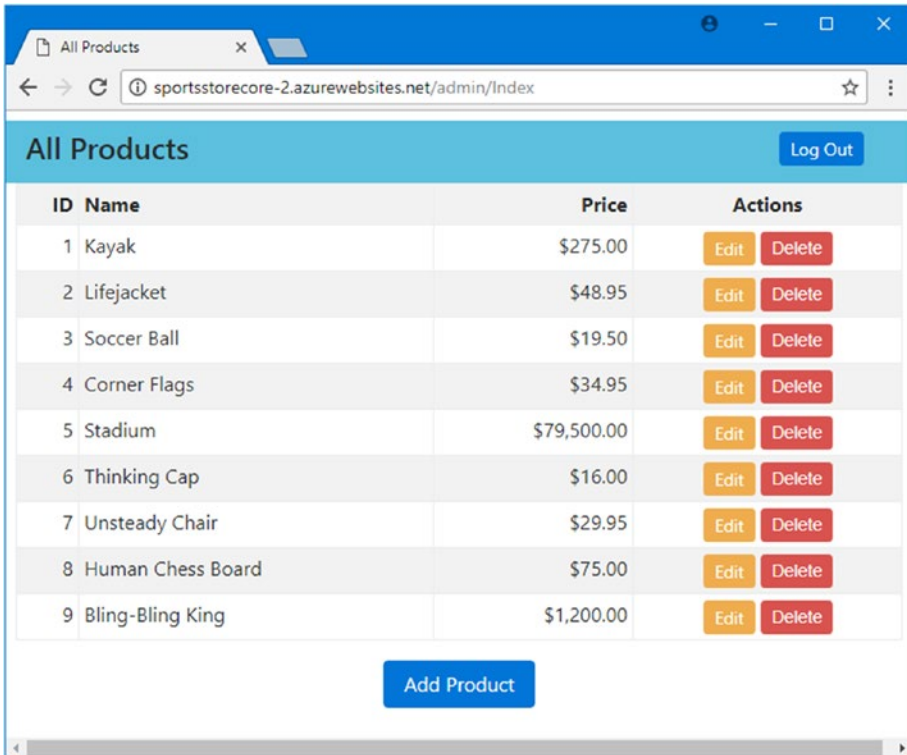


Figure 12-7. The populated database

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.