CHAPTER 11

SportsStore: Administration

In this chapter, I continue to build the SportsStore application in order 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

```
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 makes 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 the Package Manager Console and run the following commands to create a new migration and apply it to the database:

```
Add-Migration ShippedOrders
Update-Database
```

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

```csharp
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 ViewResult List() =>
            View(repository.Orders.Where(o => !o.Shipped));

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

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

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

```
        } else {
            return View(order);
        }
    }

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

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 order 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 its **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><th></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 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="panel panel-default">
    <div class="panel-heading"><h4>@ViewBag.Title</h4></div>
    <div class="panel-body">
        @RenderBody()
    </div>
</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.
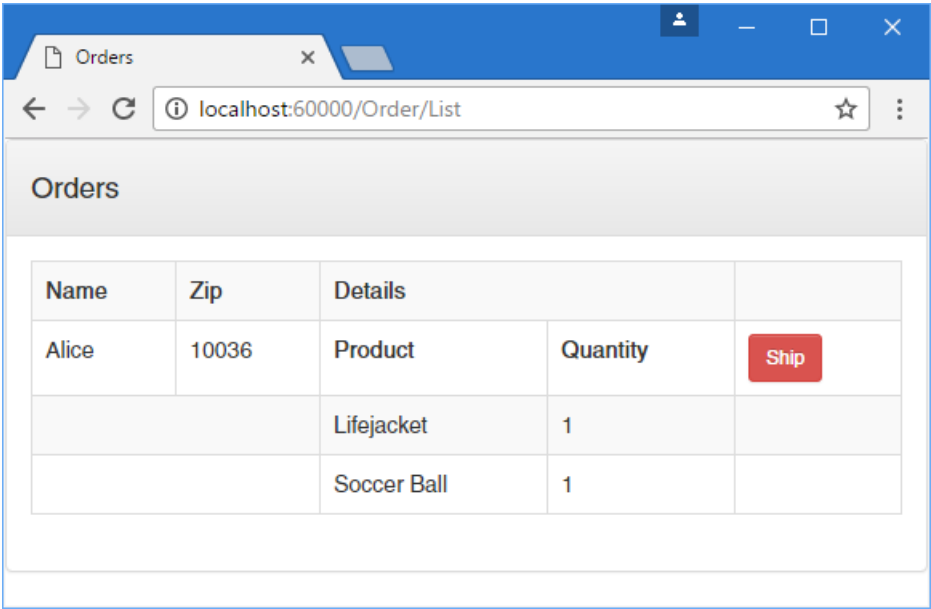

Figure 11-1. Managing orders

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.

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 used added the code shown in Listing 11-5.

Listing 11-5. The Contents of the AdminController.cs File

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

namespace SportsStore.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

        public ViewResult 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 **Admin** controller 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:

```csharp
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"},
            });

            // 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>(IActionResult 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-condensed">
    <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>
```
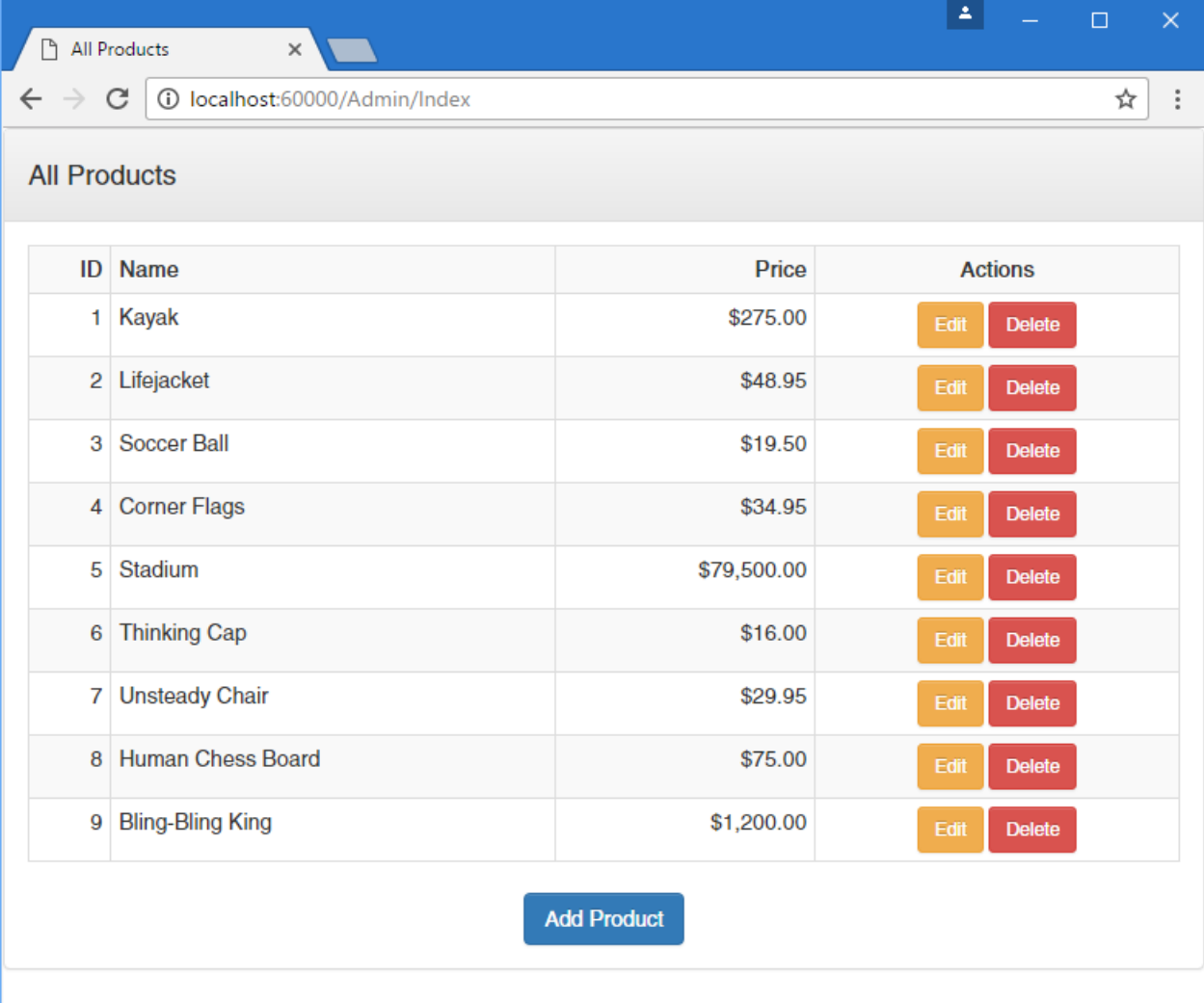
This view contains a table that has a row for each product with cells that contains the name of the table, the price, and buttons that will allow the product to be edited or deleted by sending requests to `Edit` and `Delete` actions. In addition to the table, there is an Add Product button that targets 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 in order 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.

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 to 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

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

        public ViewResult 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"},
    });

    // 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"},
    });

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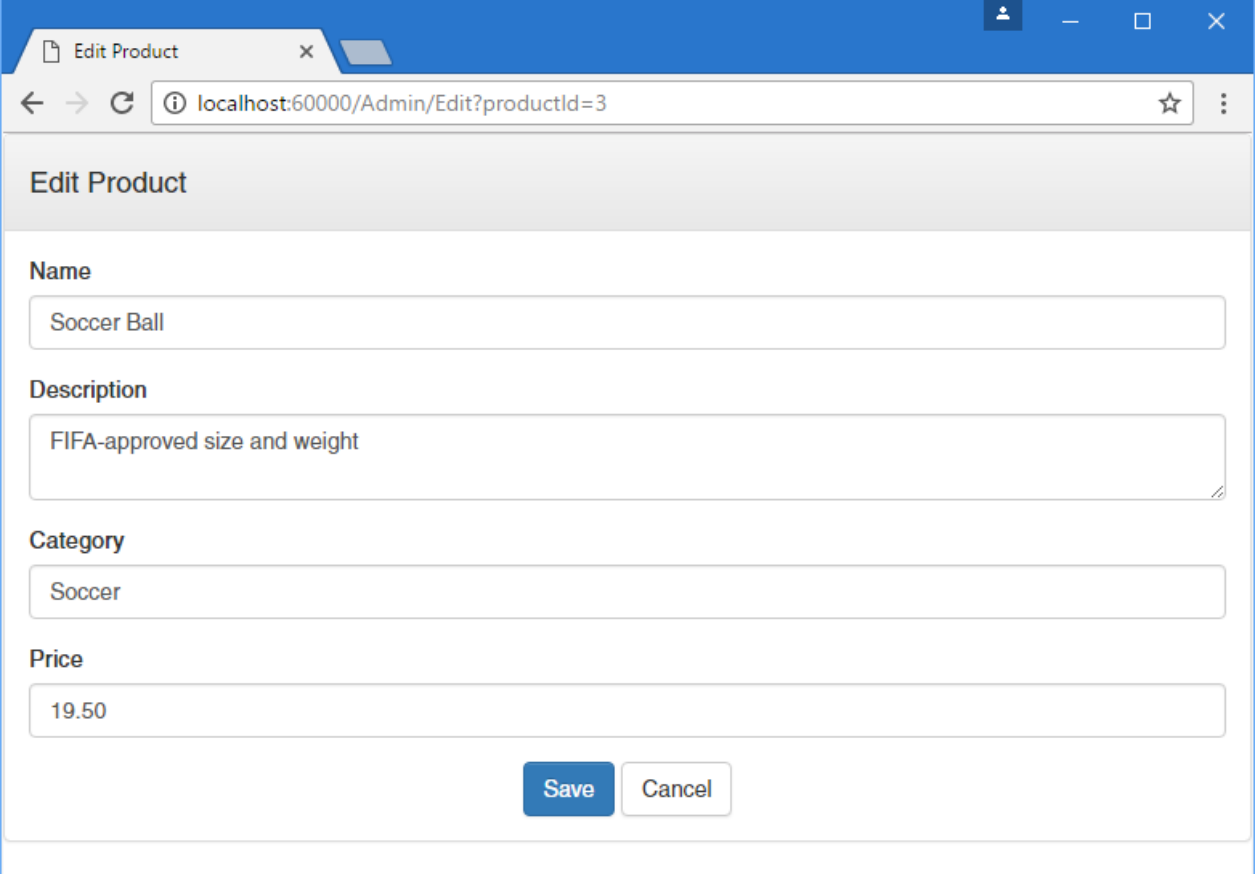
```
@model Product
@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<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-default">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 the Entity Framework Core, and safely changing it can be a complex process.

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 IProductRespository.cs File

```
using System.Collections.Generic;

namespace SportsStore.Models {

    public interface IProductRepository {
        IEnumerable<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 SaveProduct Method in the EFProductRepository.cs File

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

namespace SportsStore.Models {

    public class EFProductRepository : IProductRepository {
        private ApplicationDbContext context;

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

        public IEnumerable<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 the 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 the 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 the 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 the 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. Disconnecting a Class from an Interface in the FakeProductRepository.cs File

```
using System.Collections.Generic;

namespace SportsStore.Models {

    public class FakeProductRepository /* : IProductRepository */ {

        public IEnumerable<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 }
        };
    }
}
```

Handling Edit POST Requests

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

Listing 11-12. Defining the POST-Handling Edit Action Method in the AdminController.cs File

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

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

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

I check that the model binding process has been able to validate the data submitted to 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 that 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="panel panel-default">
    <div class="panel-heading"><h4>@ViewBag.Title</h4></div>
    <div class="panel-body">
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]</div>
        }
```

```
        @RenderBody()
    </div>
</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.
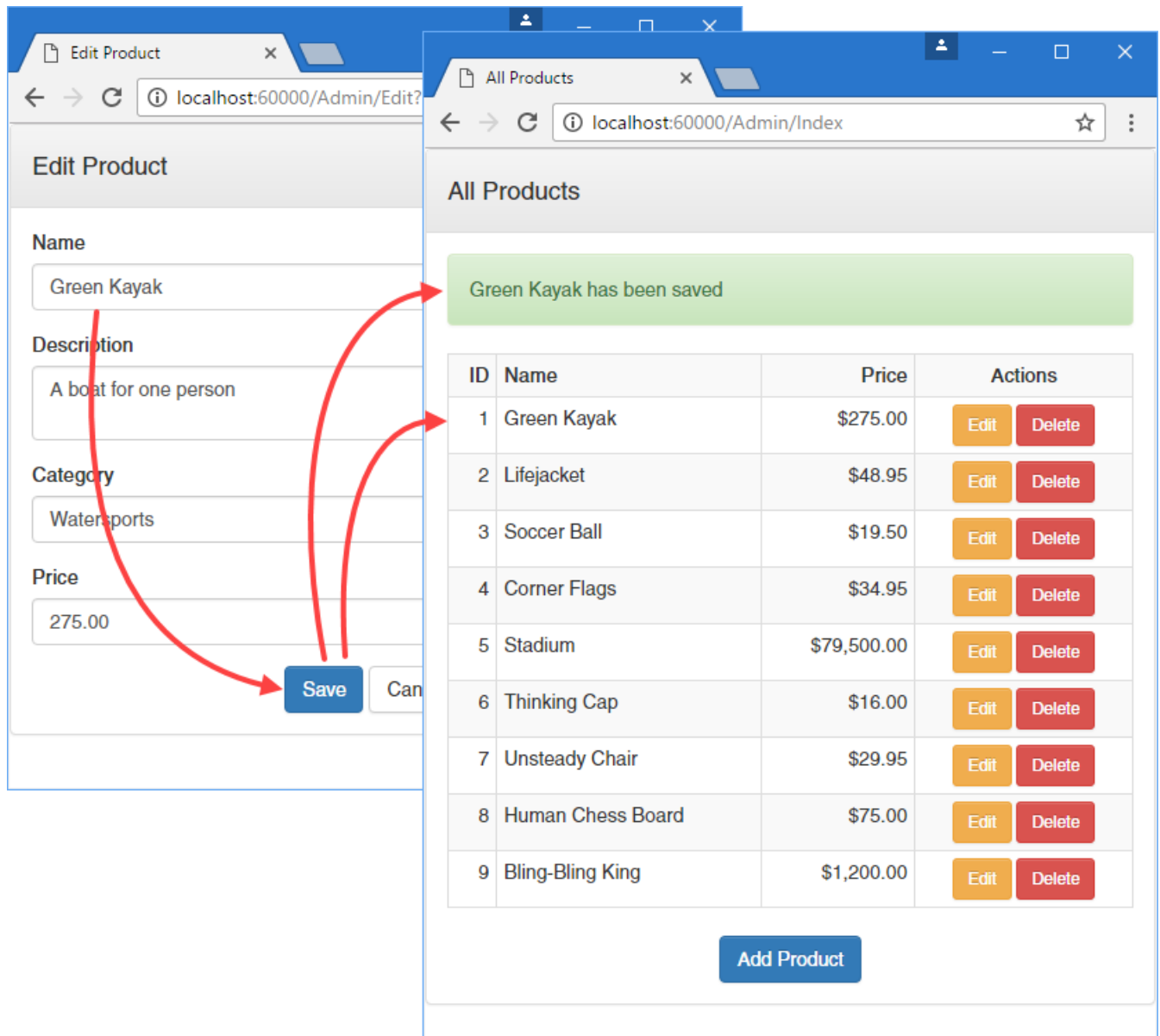
Figure 11-5. Editing a product and seeing the TempData message

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

```
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

```
@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-default">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

```
<!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="panel panel-default">
    <div class="panel-heading"><h4>@ViewBag.Title</h4></div>
    <div class="panel-body">
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>
```
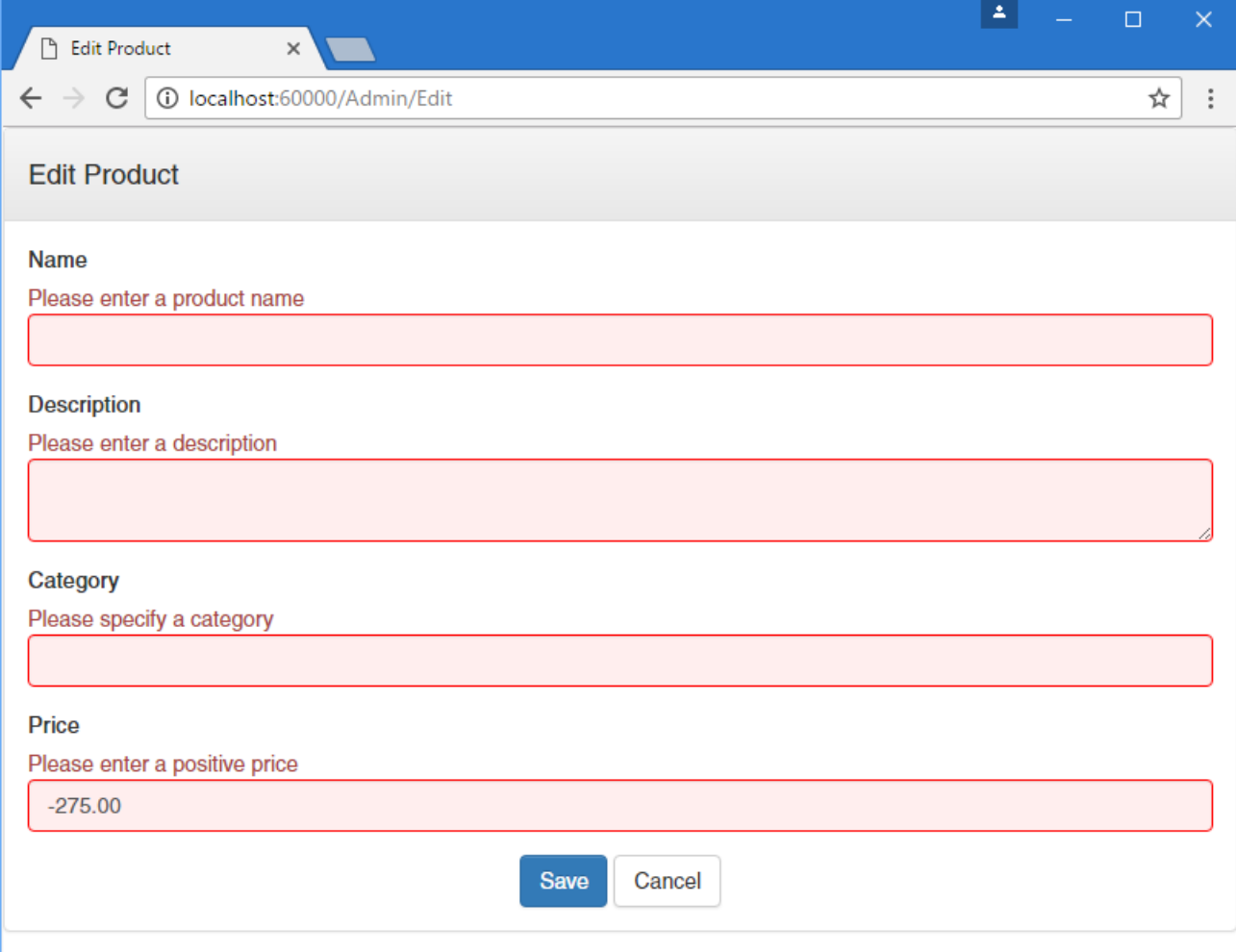
The CSS style I defined selects elements that are members of the `input-validation-error` class and applies a red border and background color.

⬚⬚**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.

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.

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. You may need to select the SportsStore project and click the Show All Items button at the top of the Solution Explorer to reveal the `bower.json` file.

Note   The client-side validation packages will not be installed correctly unless you replace the Visual Studio `git` tool, as described in Chapter 2.

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

```json
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "fontawesome": "4.6.3",
    "jquery": "2.2.4",
    "jquery-validation": "1.15.0",
    "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 Client-Side Validation Libraries to the _AdminLayout.cshtml File

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

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

These additions use a tag helper to select the files that are included in the `script` elements. I describe how this process works in Chapter 25, and this approach allows me to use wildcards to select JavaScript files, which means that the application won't break if the names of the files in the Bower package change when a new version is released. Some caution is required, however, because (as I explain in Chapter 25) it is easy to select files that you didn't expect.

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 Method to the AdminController.cs File

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

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

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

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

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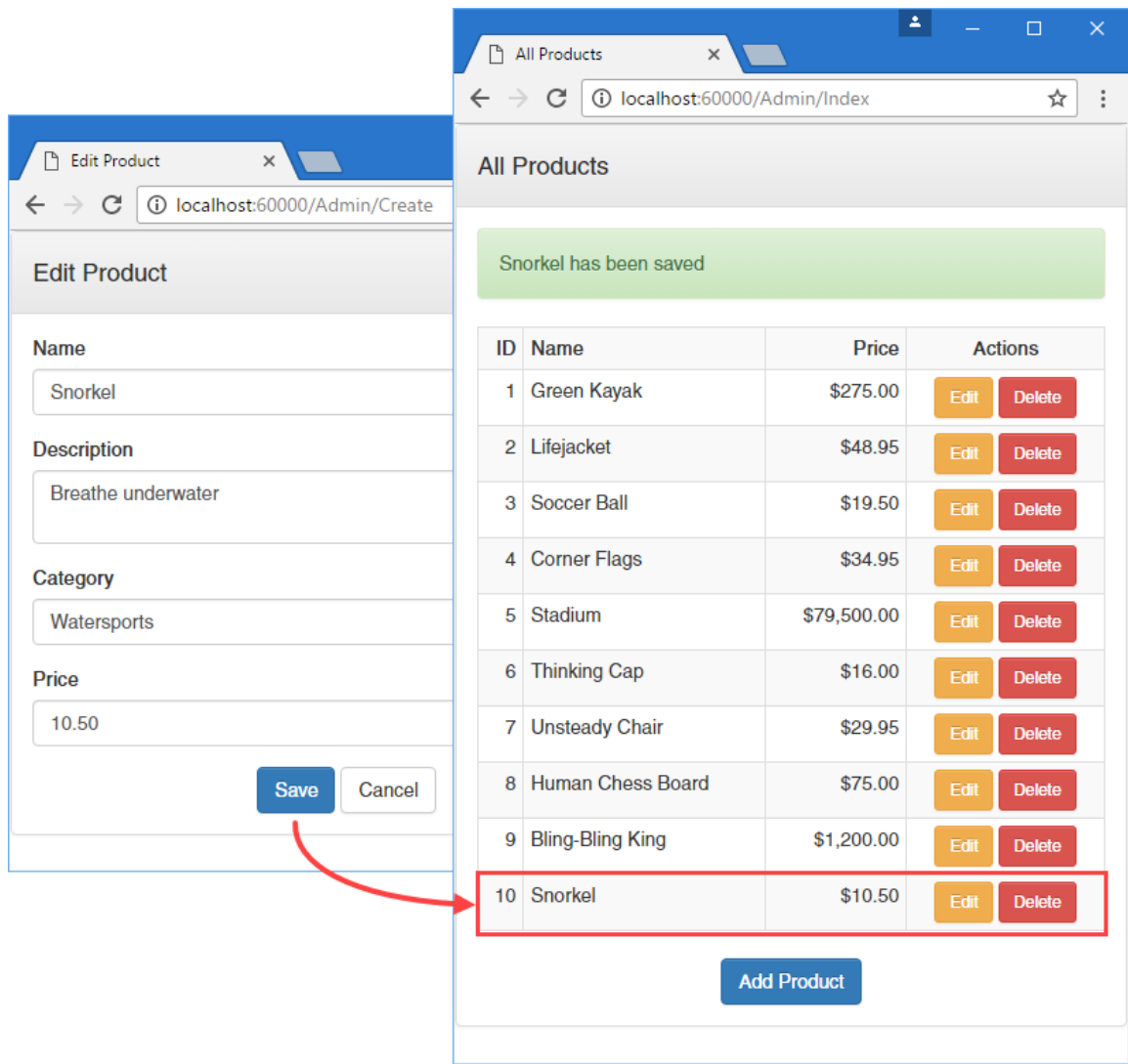
Figure 11-7. Adding a new product to the catalog

Deleting Products

Adding support for deleting items is also simple. First, I 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

```
using System.Collections.Generic;

namespace SportsStore.Models {

    public interface IProductRepository {
        IEnumerable<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

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

namespace SportsStore.Models {

    public class EFProductRepository : IProductRepository {
        private ApplicationDbContext context;

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

        public IEnumerable<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

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

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

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

```
            return View(product);
        }
    }

    public 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"},
    });

    // 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.
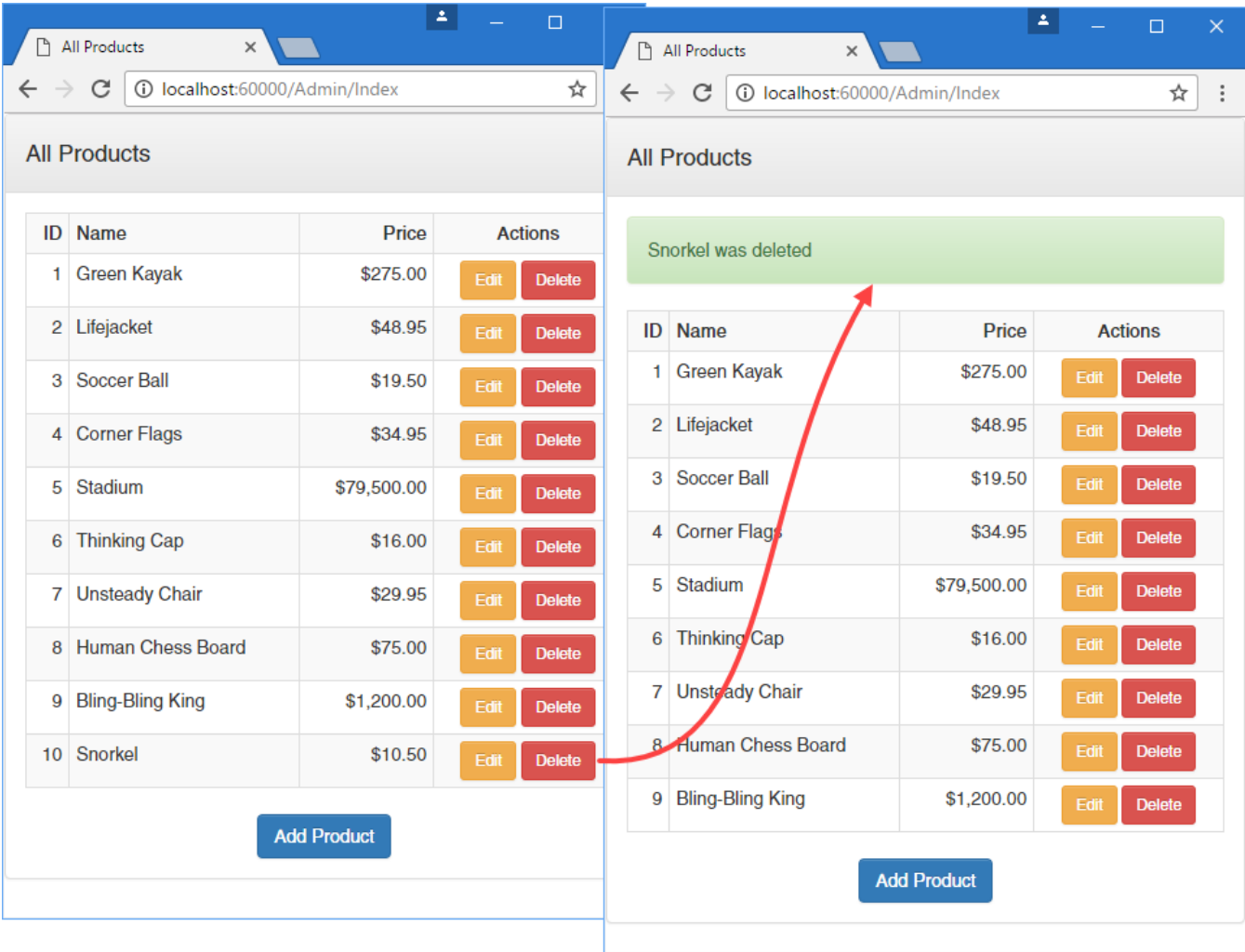

Figure 11-8. Deleting a product from the catalog

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.