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 the product catalog. I will add support for creating, editing, and removing items from the product repository, as well as for uploading and displaying images alongside products in the catalog.

Adding Catalog Management

The convention for managing collections of items is to present the user with two types of pages: a *list* page and an *edit* page, as shown in Figure 11-1.

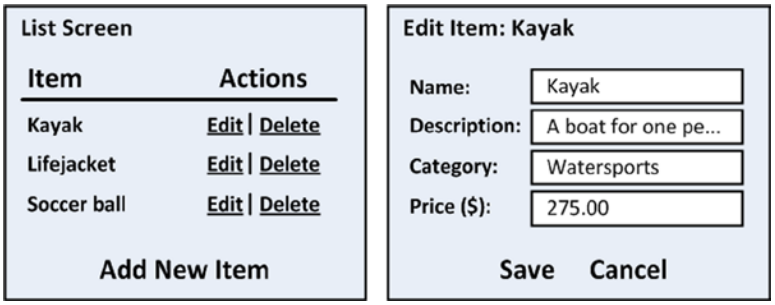


Figure 11-1. *Sketch of a CRUD UI for the product catalog*

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 tries to help by offering to generate MVC controllers that have action methods for CRUD operations and view templates that support them. But like all of the Visual Studio templates, I think it is better to learn how to use the features of the MVC Framework directly.

Creating a CRUD Controller

I am going to create a new controller for the SportsStore administration features. Right-click on the `Controllers` folder for the `SportsStore.WebUI` project in the Solution Explorer and select `Add` ➤ `Controller` from the pop-up menu. Select `MVC 5 Controller - Empty` from the list of options, click the `Add` button, set the name to `AdminController` and click the `Add` button to create the `Controllers/AdminController.cs` file. Edit the contents of the controller class to match Listing 11-1.

Listing 11-1. The Contents of the AdminController.cs File

```
using System.Web.Mvc;
using SportsStore.Domain.Abstract;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

        public ViewResult Index() {
            return View(repository.Products);
        }
    }
}
```

The controller constructor declares a dependency on the `IProductRepository` interface, which Ninject will resolve 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 `AdminTests.cs` in the `SportsStore.UnitTests` project:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

namespace SportsStore.UnitTests {
    [TestClass]
    public class AdminTests {

        [TestMethod]
        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 = ((IEnumerable<Product>)target.Index().
                ViewData.Model).ToArray();

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

Creating a New Layout

I am going to create a new layout to use with the SportsStore administration views. It will be a simple layout that provides a single point where I can apply changes to all the administration views.

Create the new layout, by right-clicking the `Views/Shared` folder in the `SportsStore.WebUI` project and select `Add > MVC 5 Layout Page (Razor)` from the pop-up menu. Set the name to `_AdminLayout.cshtml` (don't forget the underscore) and click the `OK` button to create the `Views/Shared/_AdminLayout.cshtml` file. Set the contents of the new view to match Listing 11-2.

■ **Note** As I explained previously, the convention is to start the layout name with an underscore (_). Razor is also used by another Microsoft technology called WebMatrix, which uses the underscore to prevent layout pages from being served to browsers. MVC does not need this protection, but the convention for naming layouts is carried over to MVC applications anyway.

Listing 11-2. The Contents of the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
```

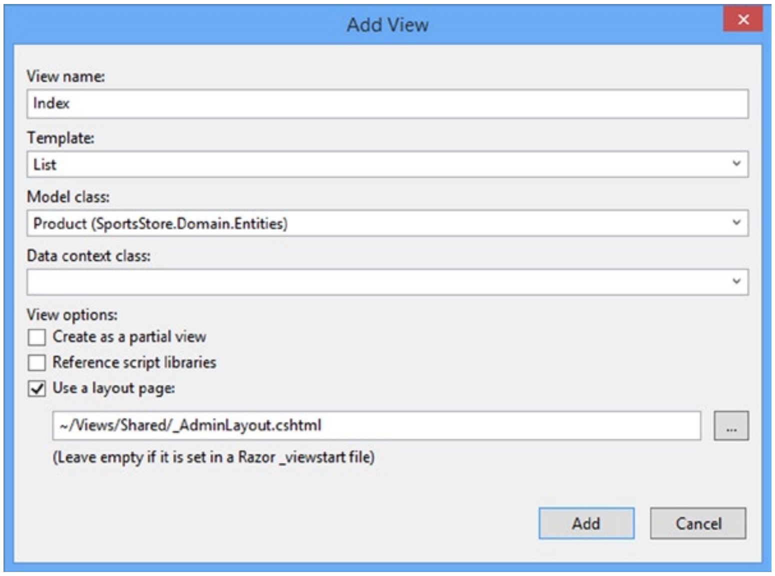
```
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

I have added a call to the RenderBody method, so that the contents of the view that the layout is being used for will be inserted into the response to the server. (I would not have had to do this if I had used the Add ➤ New Item menu option and used the Visual Studio layout template, but I took a shortcut to create the view directly, which meant I had to edit the new file to get the content I require.) I also added link elements for the Bootstrap files and for the CSS file that contains the styles I created to highlight validation errors to the user.

Implementing the List View

Now that I have the new layout, I can add a view to the project for the Index action method of the Admin controller. Even though I am not a fan of the Visual Studio scaffold and template features, I am going to create the view for the Index method using the scaffold system so you can see how it works. Just because I don't like pre-cut code, doesn't mean that you shouldn't use it.

Right-click on the Views/Admin folder in the SportsStore.WebUI project and select Add ➤ View from the menu. Set View Name to Index, select the List for the Template option (this is where I usually select one of the Empty options), select Product as the Model Class, check the option to use a layout page, and select the _AdminLayout.cshtml file from the Views/Shared folder. You can see all of the configuration options in Figure 11-2.



View name: Index
Template: List
Model class: Product (SportsStore.Domain.Entities)
Data context class:
View options:
☐ Create as a partial view
☐ Reference script libraries
☑ Use a layout page:
~/Views/Shared/_AdminLayout.cshtml
(Leave empty if it is set in a Razor _viewstart file)

Figure 11-2. Configuring a scaffold view

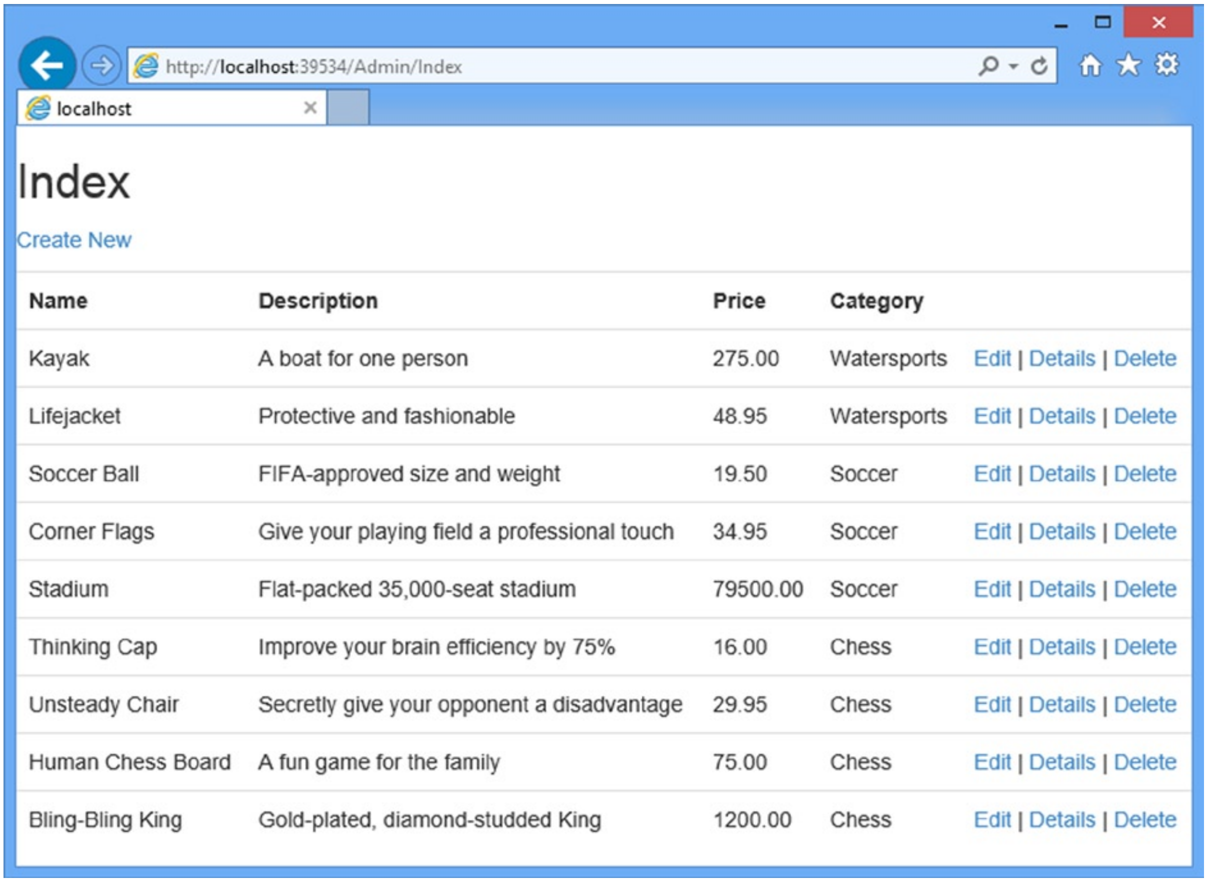
■ **Note** When using the `List` scaffold, Visual Studio assumes you are working with an `IEnumerable` sequence of the model view type, so you can just select the singular form of the class from the list.

Click the Add button to create the new view, which will have the contents shown in Listing 11-3. (I have formatted the markup so that it doesn't require so much space on the page.)

Listing 11-3. The Contents of the Views/Admin/Index.cshtml File

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>@Html.DisplayNameFor(model => model.Name)</th>
        <th>@Html.DisplayNameFor(model => model.Description)</th>
        <th>@Html.DisplayNameFor(model => model.Price)</th>
        <th>@Html.DisplayNameFor(model => model.Category)</th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>@Html.DisplayFor(modelItem => item.Name)</td>
        <td>@Html.DisplayFor(modelItem => item.Description)</td>
        <td>@Html.DisplayFor(modelItem => item.Price)</td>
        <td>@Html.DisplayFor(modelItem => item.Category)</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ProductID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ProductID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ProductID })
        </td>
    </tr>
}

</table>
```

Visual Studio looks at the type of view model object and generates elements in a table that correspond to the properties defined by the model type. You can see how this view is rendered by starting the application and navigating to the `/Admin/Index` URL. Figure 11-3 shows the results.



Index

Create New

Name	Description	Price	Category	
Kayak	A boat for one person	275.00	Watersports	Edit | Details | Delete
Lifejacket	Protective and fashionable	48.95	Watersports	Edit | Details | Delete
Soccer Ball	FIFA-approved size and weight	19.50	Soccer	Edit | Details | Delete
Corner Flags	Give your playing field a professional touch	34.95	Soccer	Edit | Details | Delete
Stadium	Flat-packed 35,000-seat stadium	79500.00	Soccer	Edit | Details | Delete
Thinking Cap	Improve your brain efficiency by 75%	16.00	Chess	Edit | Details | Delete
Unsteady Chair	Secretly give your opponent a disadvantage	29.95	Chess	Edit | Details | Delete
Human Chess Board	A fun game for the family	75.00	Chess	Edit | Details | Delete
Bling-Bling King	Gold-plated, diamond-studded King	1200.00	Chess	Edit | Details | Delete

Figure 11-3. *Rendering the scaffold List view*

The scaffold view makes a reasonable attempt at setting up a sensible baseline for the view. I have columns for each of the properties in the `Product` class and links for the other CRUD operations that target action methods in the `Admin` controller (although, since I created that controller without using scaffolding, the action methods do not exist).

The scaffolding is clever, but the views that it generates are bland and so general as to be worthless in a project of any complexity. My advice is to start with empty controllers, views and layouts and add the functionality you need as and when you need it.

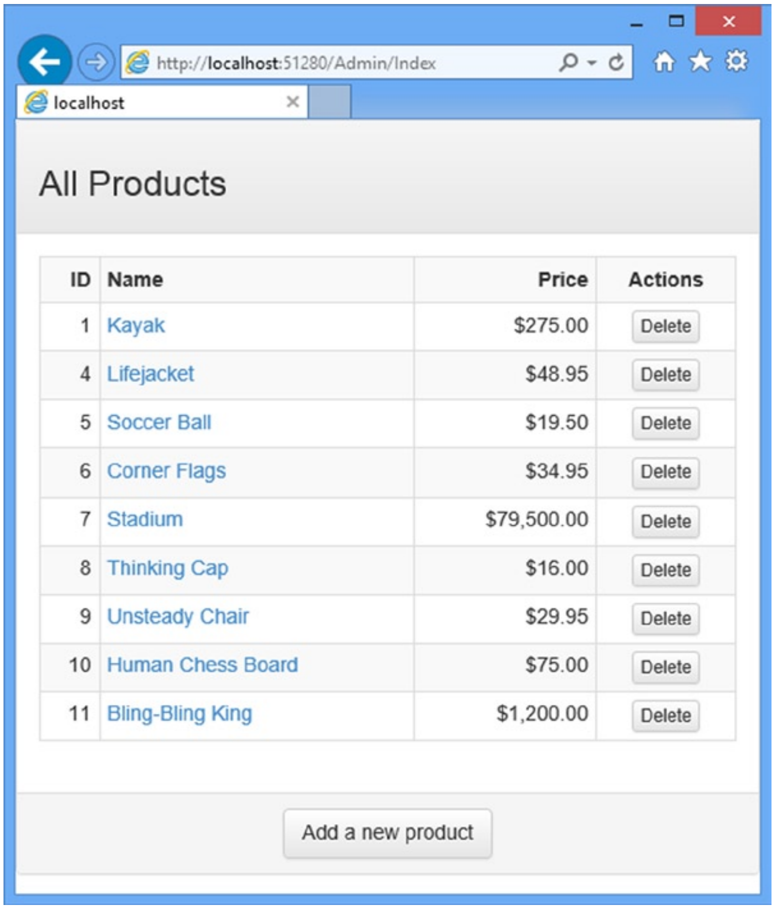
Returning to the do-it-yourself approach, edit the `Index.cshtml` file so that it corresponds to Listing 11-4.

Listing 11-4. Modifying the Index.cshtml View

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

@{
    ViewBag.Title = "Admin: All Products";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}
```

```
<div class="panel panel-default">

    <div class="panel-heading">
        <h3>All Products</h3>
    </div>
    <div class="panel-body">
        <table class="table table-striped table-condensed table-bordered">
            <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>@Html.ActionLink(item.Name, "Edit", new { item.ProductID })</td>
                    <td class="text-right">@item.Price.ToString("c")</td>
                    <td class="text-center">
                        @using (Html.BeginForm("Delete", "Admin")) {
                            @Html.Hidden("ProductID", item.ProductID)
                            <input type="submit"
                                   class="btn btn-default btn-xs"
                                   value="Delete" />
                        }
                    </td>
                </tr>
            }
        </table>
    </div>
    <div class="panel-footer">
        @Html.ActionLink("Add a new product", "Create", null,
            new { @class = "btn btn-default" })
    </div>
</div>
```

This view presents the information in a more compact form, omitting some of the properties from the Product class and using Bootstrap to apply styling. You can see how this view renders in Figure 11-4.



All Products

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

***Figure 11-4.** Rendering the modified Index view*

Now I have a nice list page. The administrator can see the products in the catalog and there are links or buttons to add, delete, and inspect items. In the following sections, I will add the functionality to support each of these actions.

Editing Products

To provide create and update features, I will add a product-editing page similar to the one shown in Figure 11-1. There are 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 11-5 shows the `Edit` method I added to the `Admin` controller. This is the action method I specified in the calls to the `Html.ActionLink` helper method in the `Index` view.

Listing 11-5. Adding the Edit Action Method in the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

        public ViewResult Index() {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }
    }
}
```

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. Obviously, I want to make sure that I am editing the product I expected. The second behavior 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 AdminTests.cs unit test file:

```
...
[TestMethod]
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 = target.Edit(1).ViewData.Model as Product;
        Product p2 = target.Edit(2).ViewData.Model as Product;
        Product p3 = target.Edit(3).ViewData.Model as Product;

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

    [TestMethod]
    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 = (Product)target.Edit(4).ViewData.Model;

        // Assert
        Assert.IsNull(result);
    }
    ...
```

Creating the Edit View

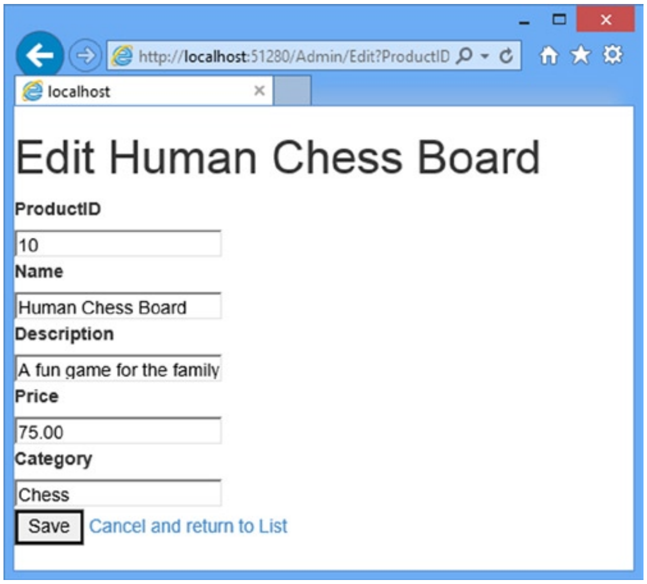
Now that I have an action method, I can create a view for it to render. Right-click on the `Views/Admin` folder in the Solution Explorer and select Add ➤ `MVC 5 View Page (Razor)` from the menu. Set the name to `Edit.cshtml`, click the button to create the file and edit the contents to match Listing 11-6.

Listing 11-6. The Contents of the Edit .cshtml File

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}
```

```
<h1>Edit @Model.Name</h1>

@using (Html.BeginForm()) {
    @Html.EditorForModel()
    <input type="submit" value="Save" />
    @Html.ActionLink("Cancel and return to List", "Index")
}
```

Instead of writing out markup for each of the labels and inputs by hand, I have called the `Html.EditorForModel` helper method. This method asks the MVC Framework to create the editing interface for me, which it does by inspecting the model type—in this case, the `Product` class. To see the page that is generated from the `Edit` view, run the application and navigate to `/Admin/Index`. Click one of the product name links, and you will see the page shown in Figure 11-5.



Edit Human Chess Board

ProductID

10

Name

Human Chess Board

Description

A fun game for the family

Price

75.00

Category

Chess

Save Cancel and return to List

Figure 11-5. *The page generated using the EditorForModel helper method*

Let's be honest: the `EditorForModel` method is convenient, but it does not produce the most attractive results. In addition, I do not want the administrator to be able to see or edit the `ProductID` attribute, and the text box for the `Description` property is far too small.

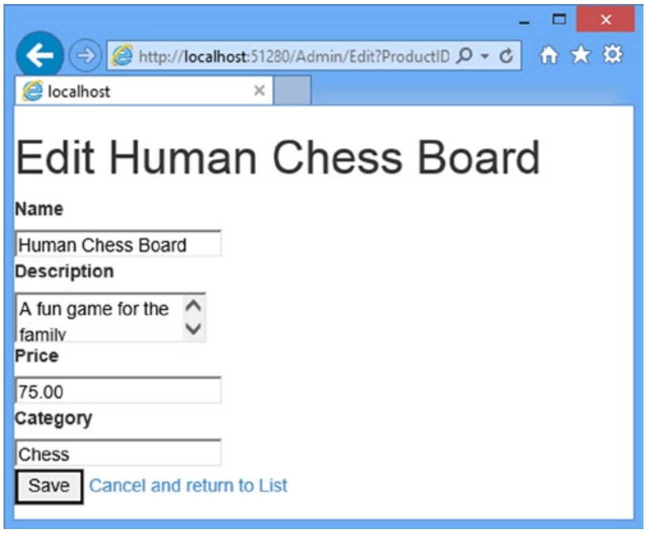
I can give the MVC Framework directions about how to create editors for properties by using *model metadata*. This allows me to apply attributes to the properties of the new model class to influence the output of the `Html.EditorForModel` method. Listing 11-7 shows how to use metadata on the `Product` class in the `SportsStore.Domain` project.

Listing 11-7. Using Model Metadata in the Product.cs File

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace SportsStore.Domain.Entities {
    public class Product {

        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }
        public string Name { get; set; }

        [DataType(DataType.MultilineText)]
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

The HiddenInput attribute tells the MVC Framework to render the property as a hidden form element, and the DataType attribute allows me to specify how a value is presented and edited. In this case, I have selected the MultilineText option. The HiddenInput attribute is part of the System.Web.Mvc namespace and the DataType attribute is part of the System.ComponentModel.DataAnnotations namespace, which is why I had you add references to the assemblies for these namespace to the SportsStore.Domain project in Chapter 7.

Figure 11-6 shows the Edit page once the metadata has been applied. You can no longer see or edit the ProductId property, and there is a multiline text box for entering the description. However, the overall appearance is poor.


http://localhost:51280/Admin/Edit?ProductID
localhost

Edit Human Chess Board

Name

Human Chess Board

Description

A fun game for the family

Price

75.00

Category

Chess

Save
Cancel and return to List

Figure 11-6. *The effect of applying metadata*

The problem is that the Html.EditorForModel helper doesn't know anything about the Product class and generates some basic and safe HTML. There are three ways we can deal with this. The first is to define CSS styles for the content that the helper generates, which is made easier by the classes that are automatically added to the HTML elements by the MVC Framework.

If you look at the source for the page shown in Figure 11-6, you will see that the textarea element that has been created for the product description has been assigned to the text-box-multi-line CSS class:

```
<textarea class="text-box-multi-line" id="Description" name="Description">
```

Other HTML elements are assigned similar classes and you can create CSS styles for each of them. This approach works well when you are creating custom styles, but doesn't make it easy to apply predefined classes like the ones that Bootstrap defines.

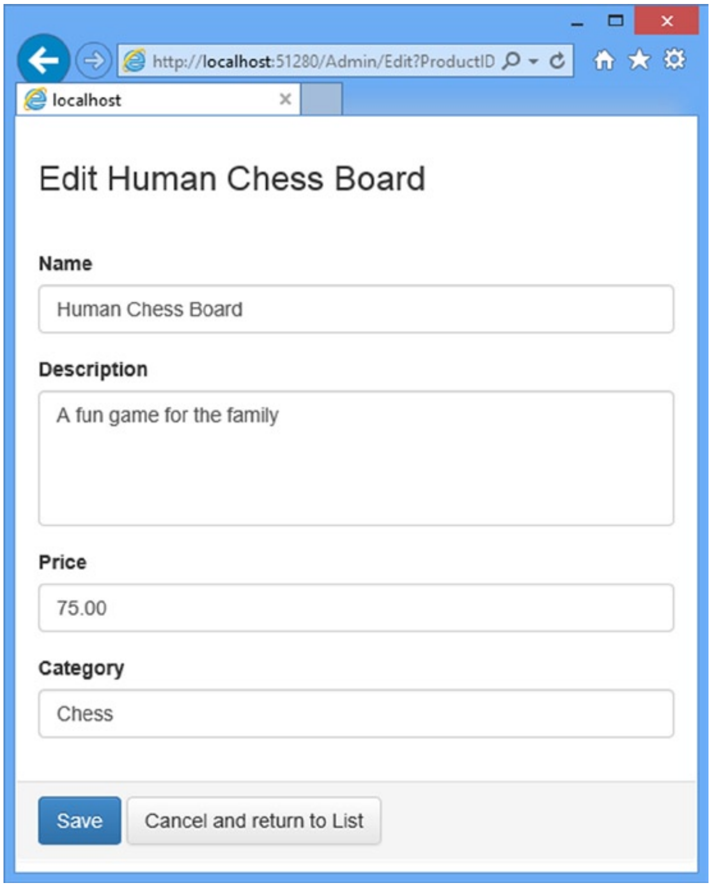
The second approach is to provide the helper with templates that it can use to generate the elements, including the styling that we require. I show you how to do this in Chapter 22.

The third approach is to create the elements needed directly, without using the model-level helper method. I like the idea of the model helper, but I rarely use it, preferring to create the HTML myself and use helpers on the individual properties, as shown in Listing 11-8.

Listing 11-8. Updating the Edit.cshtml File

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3>Edit @Model.Name</h3>
    </div>

    @using (Html.BeginForm()) {
        <div class="panel-body">
        @Html.HiddenFor(m => m.ProductID)
        @foreach (var property in ViewData.ModelMetadata.Properties) {
            if (property.PropertyName != "ProductID") {
                <div class="form-group">
                    <label>@(property.DisplayName ?? property.PropertyName)</label>
                    @if (property.PropertyName == "Description") {
                        @Html.TextArea(property.PropertyName, null,
                                new { @class = "form-control", rows = 5 })
                    } else {
                        @Html.TextBox(property.PropertyName, null,
                                new { @class = "form-control" })
                    }
                </div>
            }
        }
        </div>

        <div class="panel-footer">
            <input type="submit" value="Save" class="btn btn-primary"/>
            @Html.ActionLink("Cancel and return to List", "Index", null, new {
                @class = "btn btn-default"
            })
        </div>
    }
</div>
```

This is a variation on the metadata technique that I used in Chapter 9 and it is something that I find myself using often, even though I could achieve similar results through the HTML helper methods with the customization techniques I describe in Chapter 22. There is something pleasing about this approach that gels with my development style but, as ever with the MVC Framework, there are different approaches you can use if processing metadata doesn't do it for you. You can see how this view is displayed in the browser in Figure 11-7.


http://localhost:51280/Admin/Edit?ProductID
localhost

Edit Human Chess Board

Name

Human Chess Board

Description

A fun game for the family

Price

75.00

Category

Chess

Save
Cancel and return to List

Figure 11-7. *Displaying the editor page for products*

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 will add a new method to the `IProductRepository` interface, as shown in Listing 11-9. (As a reminder, you will find this interface in the `Abstract` folder of the `SportsStore.Domain` project.)

Listing 11-9. Adding a Method to the IProductRespository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;
```

```
namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }

        void SaveProduct(Product product);
    }
}
```

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

Listing 11-10. Implementing the SaveProduct Method in the EFProductRepository.cs File

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }

        public void SaveProduct(Product product) {

            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                Product dbEntry = context.Products.Find(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 because, as I explained earlier, it is a topic in itself and not part of the MVC Framework. 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 keeps track of the objects that it creates from the database. The object passed to the SaveChanges method is created by the MVC Framework using the default model binder, which means that the Entity Framework does not know anything about the parameter object and will not apply an update to the database. 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 *does* know about and update it explicitly.

An alternative approach would be to create a custom model binder that only obtains objects from the repository. This may seem like a more elegant approach, but it would require me to add a find capability to the repository interface so I could locate Product objects by ProductID values.

Handling Edit POST Requests

At this point, 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. The new method is shown in Listing 11-11.

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

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

        public ViewResult Index() {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }

        [HttpPost]
        public ActionResult Edit(Product product) {
            if (ModelState.IsValid) {
                repository.SaveProduct(product);
                TempData["message"] = string.Format("{0} has been saved", product.Name);
                return RedirectToAction("Index");
            } else {
                // there is something wrong with the data values
                return View(product);
            }
        }
    }
}
```

I check that the model binder 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 invoke the `Index` action method to return the user to the list of products. If there is a problem with the data, I render the `Edit` view again so that the user can make corrections.

After I have saved the changes in the repository, I store a message using the *TempData* 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 is deleted at the end of the HTTP request.

Notice that I return the `ActionResult` type from the `Edit` method. I have been using the `ViewResult` type until now. `ViewResult` is derived from `ActionResult`, and it is used when you want the framework to render a view. However, other types of `ActionResults` are available, and one of them is returned by the `RedirectToAction` method, which redirects the browser so that the `Index` action method is invoked. I describe the set of action results in Chapter 17.

I cannot use `ViewBag` in this situation because the user is being redirected. `ViewBag` passes data between the controller and view, and it cannot hold data for longer than the current HTTP request. I could have used 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 TempData 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 the model binder has created are passed to the product repository to be saved. I also want to check that invalid updates (where a model error exists) are not passed to the repository. Here are the test methods:

```
...
[TestMethod]
public void Can_Save_Valid_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"};

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

    // Assert - check that the repository was called
    mock.Verify(m => m.SaveProduct(product));
    // Assert - check the method result type
    Assert.IsNotInstanceOfType(result, typeof(ViewResult));
}
```

```
    [TestMethod]
    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
        ActionResult 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.IsInstanceOfType(result, typeof(ViewResult));
    }
    ...
```

Displaying a Confirmation Message

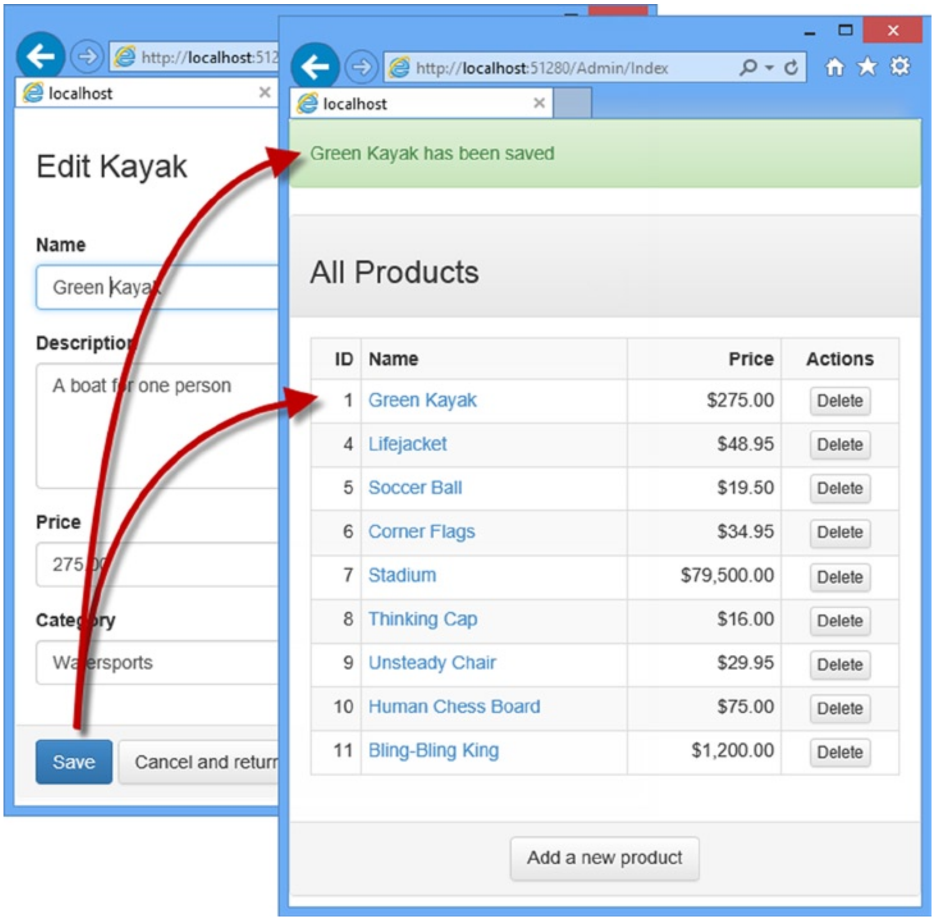
I am going to deal with the message I stored using `TempData` in the `_AdminLayout.cshtml` layout file. By handling the message in the template, I can create messages in any view that uses the template without needing to create additional Razor blocks. Listing 11-12 shows the change to the file.

Listing 11-12. Handling the ViewBag Message in the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        @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, and make some edits. Click the Save button. You will be returned to the list view, and the TempData message will be displayed, as shown in Figure 11-8.



Green Kayak has been saved

All Products

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

Add a new product

***Figure 11-8.** Editing a product and seeing the TempData message*

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.

Adding Model Validation

As is the case for most projects, I need to add validation rules to the model entity. At the moment, the administrator could enter negative prices or blank descriptions, and SportsStore would happily try and store that data in the database. Whether or not the bad data would be stored would depend on whether it conformed to the constraints in the table definition I used to create the database in Chapter 7. Listing 11-13 shows how I have applied data annotations to the Product class, just as I did for the ShippingDetails class in the last chapter.

Listing 11-13. Applying Validation Attributes to the Product.cs File

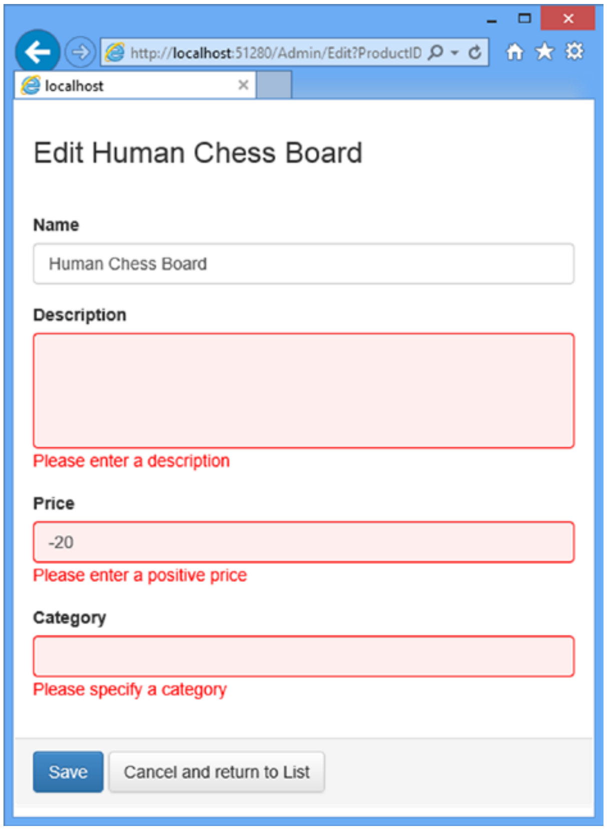
```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace SportsStore.Domain.Entities {

    public class Product {

        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

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

        [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; }
    }
}
```

The Html.TextBox and Html.TextArea helper methods that I used in the Edit.cshtml view to create the input and textarea elements will be used by the MVC Framework to signal validation problems. These signals are sent using classes for which I defined styles in the Content/ErrorStyles.css file, which have the effect of highlighting problems. I need to provide the user with details of any problems and you can see how I have done this in Listing 11-14.

Listing 11-14. Adding Validation Messages to the Edit.cshtml File

```
...
<div class="panel-body">
@foreach (var property in ViewData.ModelMetadata.Properties) {
    if (property.PropertyName != "ProductID") {
        <div class="form-group">
            <label>@(property.DisplayName ?? property.PropertyName)</label>
            @if (property.PropertyName == "Description") {
                @Html.TextArea(property.PropertyName, null,
                        new { @class = "form-control", rows = 5 })
            } else {
                @Html.TextBox(property.PropertyName, null,
                        new { @class = "form-control" })
            }
            @Html.ValidationMessage(property.PropertyName)
        </div>
    }
}
</div>
...
```

In Chapter 9, I used the `Html.ValidationSummary` helper method to create a consolidated list of all of the validation problems in the form. In this listing, I used the `Html.ValidationMessage` helper, which displays a message for a single model property. You can put the `Html.ValidationMessage` helper anywhere in a view, but it is conventional (and sensible) to put it somewhere near the element that has the validation issue to give the user some context. Figure 11-9 shows how the validation messages appear when you edit a product and enter data that breaks the rules applied to the `Product` class.


http://localhost:51280/Admin/Edit?ProductID
localhost

Edit Human Chess Board

Name: Human Chess Board

Description:

Please enter a description

Price: -20

Please enter a positive price

Category:

Please specify a category

Save
Cancel and return to List

Figure 11-9. Data validation when editing products

Enabling Client-Side Validation

At present, 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. The MVC Framework can perform client-side validation based on the data annotations I applied to the domain model class.

This feature is enabled by default, but it has not been working because I have not added links to the required JavaScript libraries. Microsoft provides support for client-side validation based on the jQuery library and a popular jQuery plug-in called, obviously enough, jQuery Validation. Microsoft extends these tools to add support for validation attributes.

The first step is to install the validation package. Select `Tools` ➤ `Library Package Manager` ➤ `Package Manager Console` in Visual Studio to open the NuGet command line and enter the following command:

```
Install-Package Microsoft.jQuery.Unobtrusive.Validation -version 3.0.0
    -projectname SportsStore.WebUI
```

■ **Tip** Do not worry if you see a message telling you that the package is already installed. Visual Studio will silently add this package to the project if you accidentally checked the `Reference Script Libraries` option when using the scaffolding feature to create a view.

Next, I need to add `script` elements to bring the JavaScript files in the package into the application HTML. The simplest place to add these links is in the `_AdminLayout.cshtml` file, so that client validation can work on any page that uses this layout. You can see the changes I made in Listing 11-15. (The order of the `script` elements is important.)

Listing 11-15. Importing JavaScript Files for Client-Side Validation into the _AdminLayout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.9.1.js"></script>
    <script src="~/Scripts/jquery.validate.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></script>
    <title></title>
</head>
<body>
    <div>
        @if (TempData["message"] != null) {
            <div class="alert alert-success">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>
```

These additions to the layout enable the client-side validation feature, providing feedback to the user about the values they are entering before they submit the form. The appearance of error messages to the user is the same because the CSS classes that are used by the server validation are also used by the client-side validation, but the response is immediate and does not require a request to be sent to the server. In most situations, client-side validation is a useful feature, but if for some reason you do not want to validate at the client, you need to add the following statements to the view:

```
...
@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
    HtmlHelper.ClientValidationEnabled = false;
    HtmlHelper.UnobtrusiveJavaScriptEnabled = false;
}
...
```

These statements disable client-side validation for the view to which they are added. You can disable client-side validation for the entire application by setting values in the `Web.config` file, like this:

```
...
<configuration>
    <appSettings>
        <add key="ClientValidationEnabled" value="false"/>
        <add key="UnobtrusiveJavaScriptEnabled" value="false"/>
    </appSettings>
</configuration>
...
```

Creating New Products

Next, I will implement the `Create` action method, which is the one specified by the `Add a new product` link in the 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 and one small change 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 11-16, to the `Admin` controller.

Listing 11-16. Adding the Create Action Method to the AdminController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
```

```
namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

        public ViewResult Create() {
            return 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 inject 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 MVC Framework ability to process the ViewResult I return as the action method result, which is something we can take for granted. (One does not usually write tests for underlying frameworks unless there is a suspicion of a problem.)

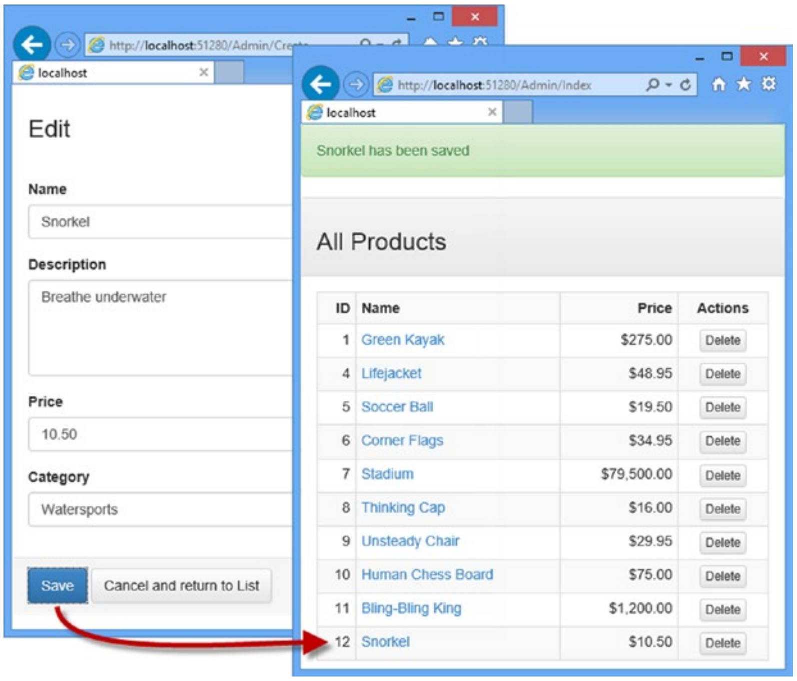
This leads to the modification. I would usually expect a form to post back to the action that rendered it, and this is what the Html.BeginForm assumes by default when it generates an HTML form. However, this does not work for the Create method, because I want the form to be posted back to the Edit action so that I can save the newly created product data. To address this, I can use an overloaded version of the Html.BeginForm helper method to specify that the target of the form generated in the Edit view is the Edit action method of the Admin controller, as shown in Listing 11-17, which illustrates the change I have made to the Views/Admin/Edit.cshtml view file.

Listing 11-17. Explicitly Specifying an Action Method and Controller for a Form in the Edit.cshtml File

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3>Edit @Model.Name</h3>
    </div>

    @using (Html.BeginForm("Edit", "Admin")) {
        <div class="panel-body">
        @foreach (var property in ViewData.ModelMetadata.Properties) {
            if (property.PropertyName != "ProductID") {
                <div class="form-group">
```

```
                    <label>@(property.DisplayName ?? property.PropertyName)</label>
                    @if (property.PropertyName == "Description") {
                        @Html.TextArea(property.PropertyName, null,
                                new { @class = "form-control", rows = 5 })
                    } else {
                        @Html.TextBox(property.PropertyName, null,
                                new { @class = "form-control" })
                    }
                    @Html.ValidationMessage(property.PropertyName)
                </div>
            }
        }
        </div>

        <div class="panel-footer">
            <input type="submit" value="Save" class="btn btn-primary"/>
            @Html.ActionLink("Cancel and return to List", "Index", null, new {
                @class = "btn btn-default"
            })
        </div>
    }
</div>
```

Now the form will always be posted to the `Edit` action, regardless of which action rendered it. I can now create products by clicking the `Add a new product` link and filling in the details, as shown in Figure 11-10.



http://localhost:51280/Admin/Create

localhost

Edit

Name

Snorkel

Description

Breathe underwater

Price

10.50

Category

Watersports

Save | Cancel and return to List

http://localhost:51280/Admin/Index

localhost

Snorkel has been saved

All Products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Delete
4	Lifejacket	$48.95	Delete
5	Soccer Ball	$19.50	Delete
6	Corner Flags	$34.95	Delete
7	Stadium	$79,500.00	Delete
8	Thinking Cap	$16.00	Delete
9	Unsteady Chair	$29.95	Delete
10	Human Chess Board	$75.00	Delete
11	Bling-Bling King	$1,200.00	Delete
12	Snorkel	$10.50	Delete

Figure 11-10. *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 11-18.

Listing 11-18. Adding a Method to Delete Products to the IProductRepository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }

        void SaveProduct(Product product);

        Product DeleteProduct(int productID);
    }
}
```

Next, I implement this method in the Entity Framework repository class, `EFProductRepository`, as shown in Listing 11-19.

Listing 11-19. Implementing Deletion Support in the EFProductRepository.cs File

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }

        public void SaveProduct(Product product) {

            if (product.ProductID == 0) {
                context.Products.Add(product);
            } else {
                Product dbEntry = context.Products.Find(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.Find(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 will explain in Chapter 16, browsers and caches are free to make GET requests without the user's explicit consent, and so I must be careful to avoid making changes as a consequence of GET requests. Listing 11-20 shows the new action method.

Listing 11-20. The Delete Action Method in the AdminController.cs File

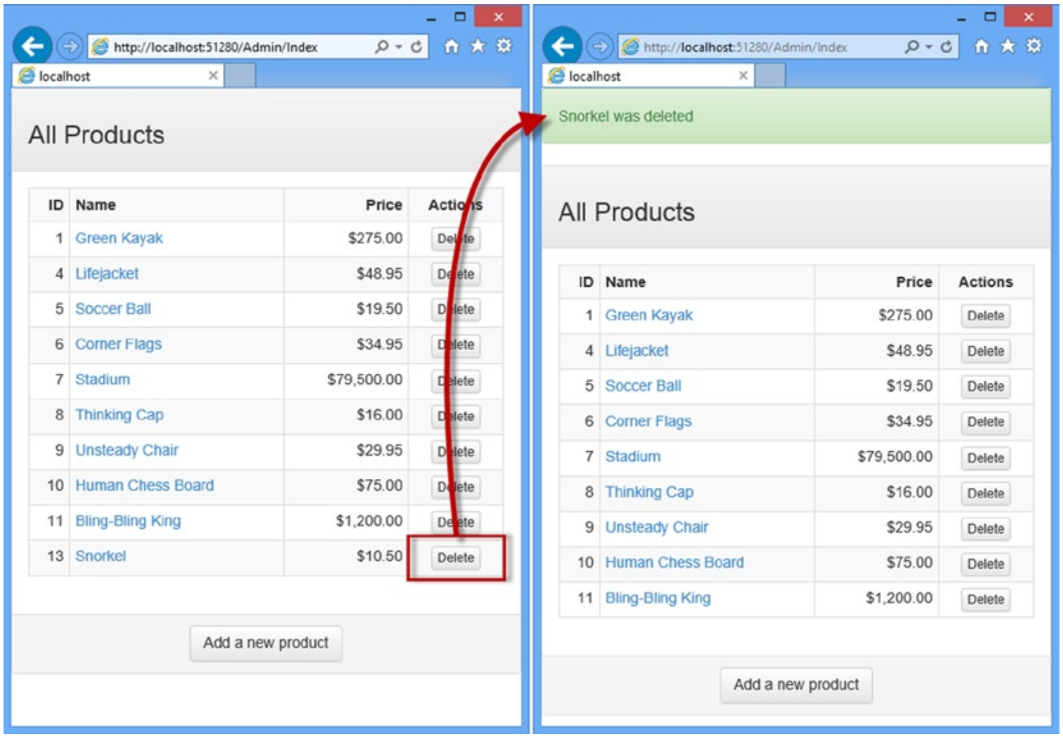
```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class AdminController : Controller {
        private IProductRepository repository;

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

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

        [HttpPost]
        public ActionResult Delete(int productId) {
            Product deletedProduct = repository.DeleteProduct(productId);
            if (deletedProduct != null) {
                TempData["message"] = string.Format("{0} was deleted",
                    deletedProduct.Name);
            }
            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:

```
...
[TestMethod]
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 clicking one of the `Delete` buttons in the product list page, as shown in Figure 11-11. 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.



All Products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Delete
4	Lifejacket	$48.95	Delete
5	Soccer Ball	$19.50	Delete
6	Corner Flags	$34.95	Delete
7	Stadium	$79,500.00	Delete
8	Thinking Cap	$16.00	Delete
9	Unsteady Chair	$29.95	Delete
10	Human Chess Board	$75.00	Delete
11	Bling-Bling King	$1,200.00	Delete
13	Snorkel	$10.50	Delete

Snorkel was deleted

All Products

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

Figure 11-11. *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. In the next chapter, I show you how to secure the administration functions so that they are not available to all users, and I add the finishing touches to complete the SportsStore functionality.