CHAPTER 12

SportsStore: Security & Finishing Touches

In the previous chapter, I added support for administering the SportsStore application, and it will not have escaped your attention that anyone could modify the product catalog if I deployed the application as it is. All they would need to know is that the administration features are available using the `Admin/Index` URL. In this chapter, I am going to show you how to prevent random people from using the administration functions by password-protecting access to the entire `Admin` controller. Once I have the security in place, I will complete the SportsStore app by adding support for product images. This may seem like a simple feature, but it requires some interesting MVC techniques.

Securing the Administration Controller

Because ASP.NET MVC is built on the core ASP.NET platform, I have access to the ASP.NET authentication and authorization features, which are packaged as a general-purpose system for keeping track of who is logged in.

DIGGING INTO THE ASP.NET SECURITY FEATURES

In this chapter, I only touch on the security features that are available. In part, this is because these features are part of the ASP.NET platform rather than the MVC Framework and in part because there are several different approaches available. I cover all the authentication and authorization features in detail in my *Pro ASP.NET MVC 5 Platform* book, which Apress will publish in 2014. But I don't want you have to buy a second book to learn about something as fundamental as web application security, and so Apress has kindly agreed to package up the key security chapters from the *Platform* book and distribute them for free from `apress.com`. They won't be available until I have written the book, of course, but it is the next project on my slate and it shouldn't be too far into 2014 before they are available for download.

Creating a Basic Security Policy

I am going to start by configuring *forms authentication*, which is one of the ways in which users can be authenticated in an ASP.NET application. In Listing 12-1, you can see the additions I have made to the `Web.config` file in the `SportsStore.WebUI` project (the one in the root of the project and not the one in the views folder).

Listing 12-1. Configuring Forms Authentication in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
  </configSections>

  <connectionStrings>
    <add name="EFDbContext" connectionString="Data Source=(localdb)\v11.0;Initial
        Catalog=SportsStore;Integrated Security=True"
            providerName="System.Data.SqlClient" />
  </connectionStrings>

  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
     <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="Email.WriteAsFile" value="true"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
    <globalization uiCulture="en-US" culture="en-US" />
    <authentication mode="Forms">
      <forms loginUrl="~/Account/Login" timeout="2880" />
    </authentication>
  </system.web>
</configuration>
```

Authentication is set up using the `authentication` element and I have used the `mode` attribute to specify that I want forms authentication, which is the kind most often used for Internet-facing web applications. In ASP.NET 4.5.1, Microsoft has added support for a wider range of Internet-suitable authentication options, which I describe in the *Pro ASP.NET MVC 5 Platform* book, as noted earlier in the chapter. I am going to stick with forms authentication because it works with local user credentials and is simple to set up and manage.

■ **Note** The main alternatives to forms authentication are *Windows authentication,* where the operating system credentials are used to identify users and *Organizational authentication*, where the user is authenticated using a cloud service such as Windows Azure. I am not going to get into either of these options because they are not widely used in Internet-facing applications.

The `loginUrl` attribute tells ASP.NET where to redirect users when they need to authenticate themselves (in this case the `~/Account/Login URL`) and the `timeout` attribute specifies how long a user remains authenticated once they have successfully logged in, expressed in minutes (2,880 minutes is 48 hours).

The next step is to tell ASP.NET where it will get details of the application users. I have broken this into a separate step because I am about to do something that should never, ever, be done in a real project: I am going to define a username and password in the `Web.config` file. You can see the changes in Listing 12-2.

Listing 12-2. Defining a Username and Password in the Web.config File

```
...
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" timeout="2880">
    <credentials passwordFormat="Clear">
      <user name="admin" password="secret" />
    </credentials>
  </forms>
</authentication>
...
```

I want to keep the example simple and focus on the way that the MVC Framework allows you to apply authentication and authorization to a web application. But putting credentials in the `Web.config` file is a recipe for disaster, especially if you set the `passwordFormat` attribute on the `credentials` element to `Clear`, meaning that passwords are stored as plain text.

■ **Caution** Don't store user credentials in the `Web.config` file and don't store passwords as plain text. See the free chapters excepted from my *Pro ASP.NET MVC 5 Platform* book (as described at the start of the section) for details of managing users via a database.

Despite being unsuitable for real projects, using the `Web.config` file to store credentials lets me focus on MVC features without getting sidetracked into aspects of the core ASP.NET platform. The result of the additions to the `Web.config` file is that I have a hard-coded username (`admin`) and password (`secret`).

Applying Authorization with Filters

The MVC Framework has a powerful feature called *filters*. These are .NET attributes that you can apply to an action method or a controller class and they introduce additional logic when a request is processed to change the behavior of the MVC Framework.

There are different kinds of filters available and you can create your own custom filters, as I explain in Chapter 18. The filter that interests me at the moment is the default *authorization filter*, `Authorize`. In Listing 12-3, you can see how I have applied this filter to the `Admin` controller.

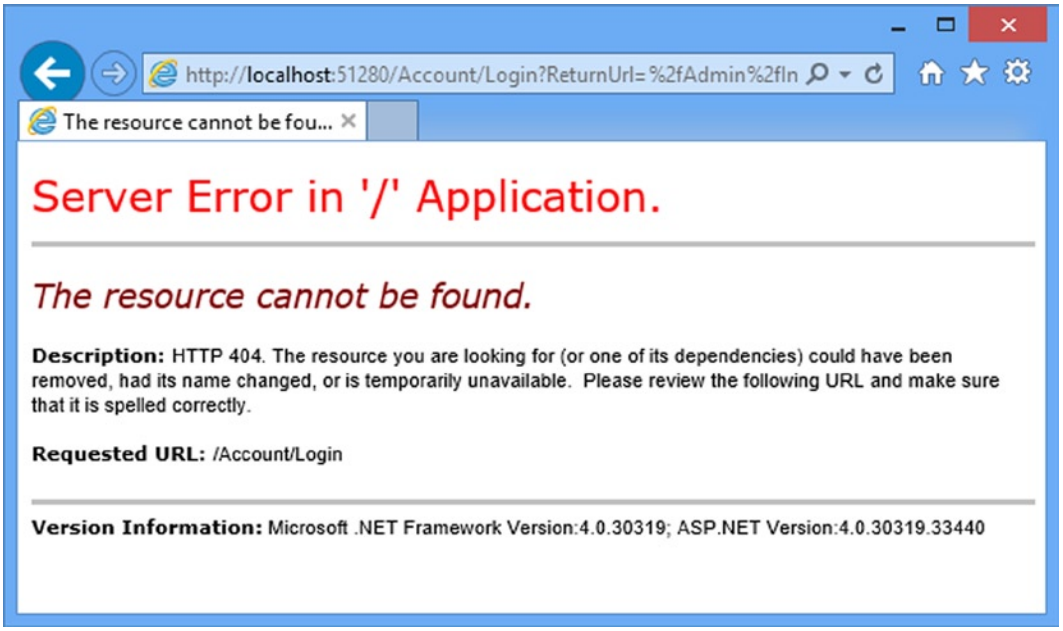
Listing 12-3. Adding the Authorize Attribute in the AdminController.cs File

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

namespace SportsStore.WebUI.Controllers {

    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;
        public AdminController(IProductRepository repo) {
            repository = repo;
        }

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

When applied without parameters, the Authorize attribute grants access to the controller action methods to all authenticated users. This means that if you are authenticated, you are automatically authorized to use the administration features. This is fine for SportsStore, where there is only one set of restricted action methods and only one user.

■ **Note** You can apply filters to an individual action method or to a controller. When you apply a filter to a controller, it works as though you had applied it to every action method in the controller class. In Listing 12-3, I applied the Authorize filter to the class, so all of the action methods in the Admin controller are available only to authenticated users.

You can see the effect that the Authorize filter has by running the application and navigating to the /Admin/Index URL. You will see an error similar to the one shown in Figure 12-1.


http://localhost:51280/Account/Login?ReturnUrl=%2fAdmin%2fIn
The resource cannot be fou...

Server Error in '/' Application.

The resource cannot be found.

Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly.

Requested URL: /Account/Login

Version Information: Microsoft .NET Framework Version:4.0.30319; ASP.NET Version:4.0.30319.33440

Figure 12-1. *The effect of the Authorize filter*

When you try to access the Index action method of the Admin controller, the MVC Framework detects the Authorize filter. Because you have not been authenticated, you are redirected to the URL specified in the Web.config forms authentication section: /Account/Login. I have not created the Account controller yet (which is what causes the error shown in the figure), but the fact that the MVC Framework has tried to redirect the request shows that the Authorize filter is working.

Creating the Authentication Provider

Using the forms authentication feature requires calls to two static methods of the System.Web.Security.FormsAuthentication class:

- The Authenticate method validates credentials supplied by the user.
- The SetAuthCookie method adds a cookie to the response to the browser, so that users do not need to authenticate every time they make a request.

The problem with calling static methods from within action methods is that it makes unit testing the controller difficult: mocking frameworks typically mock only instance members. The classes that comprise the MVC Framework have been designed with unit testing in mind, but the FormsAuthentication class predates the unit testing-friendly design of MVC.

The best way to address the problem is to decouple the controller from the static methods using an interface, which offers the additional benefit that this fits in with the broader MVC design pattern and makes it easier to switch to a different authentication system later.

I start by defining the authentication provider interface. Create a new folder called Abstract in the Infrastructure folder of the SportsStore.WebUI project and add a new interface called IAuthProvider. The contents of this interface are shown in Listing 12-4.

Listing 12-4. The Contents of the IAuthProvider.cs File

```
namespace SportsStore.WebUI.Infrastructure.Abstract {

    public interface IAuthProvider {
        bool Authenticate(string username, string password);
    }
}
```

I can now create an implementation of this interface that acts as a wrapper around the static methods of the FormsAuthentication class. Create another new folder in Infrastructure—this time called Concrete—and create a new class called FormsAuthProvider. The contents of this class are shown in Listing 12-5.

Listing 12-5. The Contents of the FormsAuthProvider.cs File

```
using System.Web.Security;
using SportsStore.WebUI.Infrastructure.Abstract;

namespace SportsStore.WebUI.Infrastructure.Concrete {

    public class FormsAuthProvider : IAuthProvider {

        public bool Authenticate(string username, string password) {
            bool result = FormsAuthentication.Authenticate(username, password);
            if (result) {
                FormsAuthentication.SetAuthCookie(username, false);
            }
            return result;
        }
    }
}
```

■ **Note** You will see a warning from Visual Studio that the `FormsAuthentication.Authenticate` method has been deprecated. This is part of Microsoft's ongoing efforts to rationalize user security, which is a thorny area for any web application framework. For this chapter, the deprecated method will suffice and allow me to perform authentication using the static details I added to the `Web.config` file.

The implementation of the `Authenticate` method calls the static `FormsAuthentication` methods that I wanted to keep out of the controller. The final step is to register the `FormsAuthProvider` in the `AddBindings` method of the `NinjectDependencyResolver` class, as shown in Listing 12-6.

Listing 12-6. Registering the Authentication Provider in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Infrastructure.Concrete;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }
        private void AddBindings() {
            kernel.Bind<IProductRepository>().To<EFProductRepository>();

            EmailSettings emailSettings = new EmailSettings {
                WriteAsFile = bool.Parse(ConfigurationManager
                    .AppSettings["Email.WriteAsFile"] ?? "false")
            };

            kernel.Bind<IOrderProcessor>().To<EmailOrderProcessor>()
                .WithConstructorArgument("settings", emailSettings);

            kernel.Bind<IAuthProvider>().To<FormsAuthProvider>();
        }
    }
}
```

Creating the Account Controller

The next task is to create the Account controller and the Login action method referred to in the Web.config file. In fact, I will create two versions of the Login method. The first will render a view that contains a login prompt, and the other will handle the POST request when users submit their credentials.

To get started, I created a view model class that I will pass between the controller and the view. Add a new class file called LoginViewModel.cs to the Models folder of the SportsStore.WebUI project and edit the content so that it matches Listing 12-7.

Listing 12-7. The Contents of the LoginViewModel.cs File

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.WebUI.Models {

    public class LoginViewModel {
        [Required]
        public string UserName { get; set; }

        [Required]
        public string Password { get; set; }
    }
}
```

This class contains properties for the username and password, and uses data annotation attributes to specify that values for both are required. Given that there are only two properties, you might be tempted to do without a view model and rely on the ViewBag to pass data to the view. However, it is good practice to define view models so that the data passed from the controller to the view and from the model binder to the action method is typed consistently.

Next, I create the Account controller that will handle authentication. Create a new class file called AccountController.cs in the Controllers folder and edit the file contents to match Listing 12-8.

Listing 12-8. The the Contents of the AccountController.cs File

```
using System.Web.Mvc;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;
```

```
namespace SportsStore.WebUI.Controllers {

    public class AccountController : Controller {
        IAuthProvider authProvider;

        public AccountController(IAuthProvider auth) {
            authProvider = auth;
        }

        public ViewResult Login() {
            return View();
        }

        [HttpPost]
        public ActionResult Login(LoginViewModel model, string returnUrl) {

            if (ModelState.IsValid) {
                if (authProvider.Authenticate(model.UserName, model.Password)) {
                    return Redirect(returnUrl ?? Url.Action("Index", "Admin"));
                } else {
                    ModelState.AddModelError("", "Incorrect username or password");
                    return View();
                }
            } else {
                return View();
            }
        }
    }
}
```

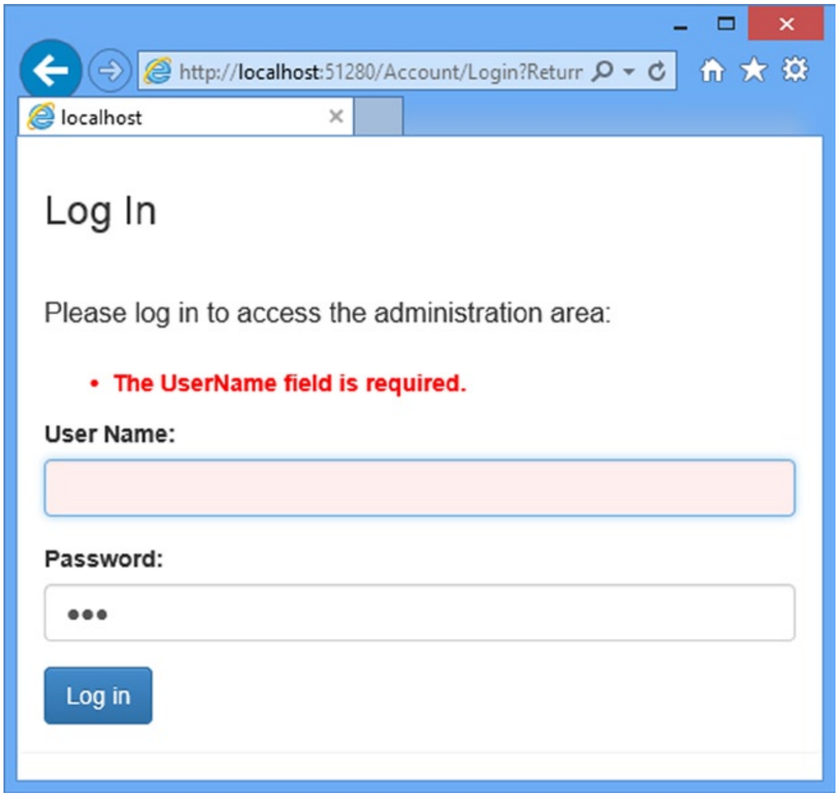
Creating the View

To create the view that will ask the users for their credentials, create the `Views/Account` folder in the `SportsStore.WebUI` folder. Right-click on the new folder, select Add ➤ `MVC 5 View Page (Razor)` from the menu, set the name to `Login` and click `OK` to create the `Login.cshtml` file. Edit the contents of the new file to match Listing 12-9.

Listing 12-9. The Contents of the Login.cshtml File

```
@model SportsStore.WebUI.Models.LoginViewModel

@{
    ViewBag.Title = "Admin: Log In";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<div class="panel">
    <div class="panel-heading">
        <h3> Log In</h3>
    </div>
    <div class="panel-body">
        <p class="lead">Please log in to access the administration area:</p>
        @using (Html.BeginForm()) {
            @Html.ValidationSummary()
            <div class="form-group">
                <label>User Name:</label>
                @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
            </div>
            <div class="form-group">
                <label>Password:</label>
                @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
            </div>
            <input type="submit" value="Log in"  class="btn btn-primary"/>
        }
    </div>
</div>
```

This view uses the `_AdminLayout.cshtml` layout and Bootstrap classes to style the content. There are no new techniques in this view, other than the use of the `Html.PasswordFor` helper method, which generates an `input` element whose `type` attribute is set to `password`. I describe the complete set of HTML helper methods in Chapter 21. You can see how the view appears by starting the app and navigating to the `/Admin/Index` URL, as shown in Figure 12-2.



Log In

Please log in to access the administration area:

- The UserName field is required.

User Name:

Password:

Figure 12-2. *The Login view*

The Required attributes that I applied to the properties of the view model are enforced using client-side validation. (Remember that the required JavaScript libraries are included in the _AdminLayout.cshtml layout created in the previous chapter.) Users can submit the form only after they have provided both a username and password, and the authentication is performed at the server when I call the FormsAuthentication.Authenticate method.

■ **Caution** In general, using client-side data validation is a good idea. It offloads some of the work from your server and gives users immediate feedback about the data they are providing. However, you should not be tempted to perform authentication at the client, as this would typically involve sending valid credentials to the client so they can be used to check the username and password that the user has entered, or at least trusting the client's report of whether they have successfully authenticated. Authentication must always be done at the server.

When I receive bad credentials, I add an error to the ModelState and re-render the view. This causes a message to be displayed in the validation summary area, which I created by calling the Html.ValidationSummary helper method in the view. This takes care of protecting the SportsStore administration functions. Users will be allowed to access these features only after they have supplied valid credentials and received a cookie, which will be attached to subsequent requests.

UNIT TEST: AUTHENTICATION

Testing the Account controller requires me to check two behaviors: a user should be authenticated when valid credentials are supplied, and a user should *not* be authenticated when invalid credentials are supplied. I perform these tests by creating mock implementations of the IAuthProvider interface and checking the type and nature of the result of the controller Login method. I created the following tests in a new unit test file called AdminSecurityTests.cs:

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;

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

        [TestMethod]
        public void Can_Login_With_Valid_Credentials() {

            // Arrange - create a mock authentication provider
            Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
            mock.Setup(m => m.Authenticate("admin", "secret")).Returns(true);

            // Arrange - create the view model
            LoginViewModel model = new LoginViewModel {
                UserName = "admin",
                Password = "secret"
            };

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

            // Act - authenticate using valid credentials
            ActionResult result = target.Login(model, "/MyURL");

            // Assert
            Assert.IsInstanceOfType(result, typeof(RedirectResult));
            Assert.AreEqual("/MyURL", ((RedirectResult)result).Url);
        }

        [TestMethod]
        public void Cannot_Login_With_Invalid_Credentials() {

            // Arrange - create a mock authentication provider
            Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
            mock.Setup(m => m.Authenticate("badUser", "badPass")).Returns(false);

            // Arrange - create the view model
            LoginViewModel model = new LoginViewModel {
                UserName = "badUser",
                Password = "badPass"
            };

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

            // Act - authenticate using valid credentials
            ActionResult result = target.Login(model, "/MyURL");

            // Assert
            Assert.IsInstanceOfType(result, typeof(ViewResult));
            Assert.IsFalse(((ViewResult)result).ViewData.ModelState.IsValid);
        }
    }
}
```

Image Uploads

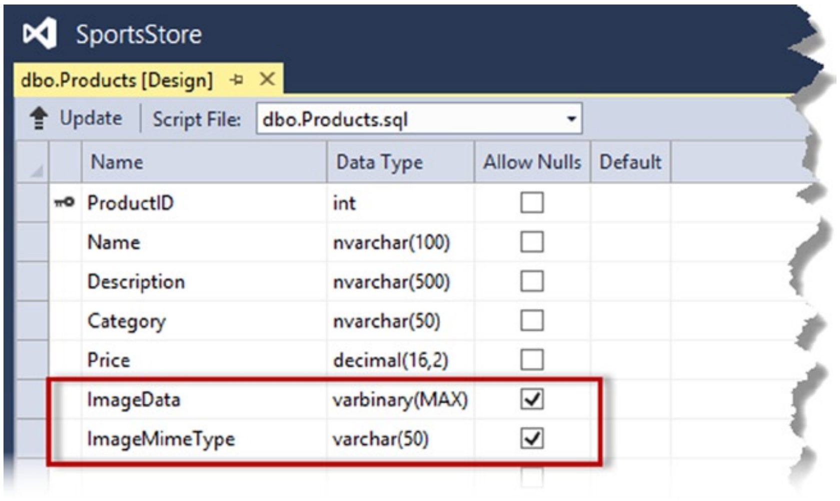
I am going to complete the SportsStore user experience with something a little more sophisticated: I will add the ability for the administrator to upload product images and store them in the database so that they are displayed in the product catalog. This isn't something that is especially interesting or useful in its own right, but it does allow me to demonstrate some important MVC Framework features.

Extending the Database

Open the Visual Studio Server Explorer window and navigate to the Products table in the database created in Chapter 7. The name of the data connection may have changed to be EFDbContext, which is the name assigned to the connection in the Web.config file. Visual Studio is a little bit inconsistent about when it renames the connection, so you might also see the original name that was shown when the connection was created. Right-click on the Products table and select New Query from the pop-up menu and enter the following SQL into the text area:

```
ALTER TABLE [dbo].[Products]
    ADD [ImageData]     VARBINARY (MAX) NULL,
        [ImageMimeType] VARCHAR (50)    NULL;
```

Click the Execute button (which is marked with an arrow) in the top-left cover of the window and Visual Studio will update the database, adding two new columns to the table. To test the update, right-click on the Products table in the Server Explorer window and select Open Table Definition from the menu. You will see that there are now columns called ImageData and ImageMimeType, as shown in Figure 12-3.



Name	Data Type	Allow Nulls	Default
ProductID	int	☐	
Name	nvarchar(100)	☐	
Description	nvarchar(500)	☐	
Category	nvarchar(50)	☐	
Price	decimal(16,2)	☐	
ImageData	varbinary(MAX)	✓	
ImageMimeType	varchar(50)	✓	

Figure 12-3. *Adding columns to the database*

■ **Tip** If the columns are not visible, close the design window, right-click on the data connection in the Server Explorer window and select Refresh from the menu. The new columns should now be visible when you select the Open Table Definition menu item again.

Enhancing the Domain Model

I need to add two new fields to the Products class in the SportsStore.Domain project that correspond to the new database columns, as shown in Listing 12-10.

Listing 12-10. Adding Properties 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; }

        [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; }

        public byte[] ImageData { get; set; }
        public string ImageMimeType { get; set; }
    }
}
```

■ **Caution** Make sure that the names of the properties that you add to the `Product` class exactly match the names you gave to the new columns in the database.

Creating the Upload User Interface Elements

The next step is to add support for handling file uploads. This involves creating a UI that the administrator can use to upload an image. Modify the `Views/Admin/Edit.cshtml` view so that it matches Listing 12-11.

Listing 12-11. Adding Support for Images 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",
        FormMethod.Post, new { enctype = "multipart/form-data" })) {

    <div class="panel-body">
        @Html.HiddenFor(m => m.ProductID)
        @foreach (var property in ViewData.ModelMetadata.Properties) {
            switch (property.PropertyName) {
                case "ProductID":
                case "ImageData":
                case "ImageMimeType":
                    // do nothing
                    break;
                default:
                    <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>
                    break;
            }
        }

        <div class="form-group">
            <div style="position:relative;">
                <label>Image</label>
                <a class='btn' href='javascript:;'>
                    Choose File...
                    <input type="file" name="Image" size="40"
                        style="position:absolute;z-index:2;top:0;
                            left:0;filter: alpha(opacity=0); opacity:0;
                            background-color:transparent;color:transparent;"
                        onchange='$("#upload-file-info").html($(this).val());'>
                </a>
                <span class='label label-info' id="upload-file-info"></span>
            </div>
            @if (Model.ImageData == null) {
                <div class="form-control-static">No Image</div>
            } else {
                <img class="img-thumbnail" width="150" height="150"
                        src="@Url.Action("GetImage", "Product",
                        new { Model.ProductID })" />
            }
        </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>
```

You may already be aware that Web browsers will upload files properly only when the HTML form element defines an enctype value of multipart/form-data. In other words, for a successful upload, the form element must look like this:

```
...
<form action="/Admin/Edit" enctype="multipart/form-data" method="post">
...
```

Without the enctype attribute, the browser will transmit only the name of the file and not its content, which is no use at all. To ensure that the enctype attribute appears, I must use an overload of the Html.BeginForm helper method that lets me specify HTML attributes, like this:

```
...
@using (Html.BeginForm("Edit", "Admin",
    FormMethod.Post, new { enctype = "multipart/form-data" })) {
...
```

There are two other changes in the view. The first is that I have replaced the Razor if expression I used when generating input elements with a switch statement. The effect is the same, but it allows me to specify the model properties I want to skip more concisely, and I don't want to display the image-related properties directly to the user.

Instead, I have made the remaining change, which is to add an input element whose type is file to allow file upload, along with an img element to display the image associated with a product, if there is one in the database.

The horrific mess of inline CSS and JavaScript addresses a shortcoming in the Bootstrap library: it does not properly style file input elements. There are a number of extensions that add the missing functionality, but I have chosen the magic incantation shown in the listing because it is self-contained and is reliable. It doesn't change the way that the MVC Framework works, just the way in which the elements in the Edit.cshtml file are styled.

Saving Images to the Database

I need to enhance the POST version of the Edit action method in the Admin controller so that I take the image data that has been uploaded and save it in the database. Listing 12-12 shows the changes that are required.

Listing 12-12. Handling Image Data in the AdminController.cs File

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

namespace SportsStore.WebUI.Controllers {

    [Authorize]
    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, HttpPostedFileBase image = null) {
            if (ModelState.IsValid) {
                if (image != null) {
                    product.ImageMimeType = image.ContentType;
                    product.ImageData = new byte[image.ContentLength];
                    image.InputStream.Read(product.ImageData, 0, image.ContentLength);
                }
                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);
            }
        }

        public ViewResult Create() {
            return View("Edit", new Product());
        }

        [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");
        }
    }
}
```

I have added a new parameter to the `Edit` method, which the MVC Framework uses to pass the uploaded file data to the action method. I check to see if the parameter value is `null`; if it is not, I copy the data and the MIME type from the parameter to the `Product` object so that it is saved to the database. I must also update the `EFProductRepository` class in the `SportsStore.Domain` project to ensure that the values assigned to the `ImageData` and `ImageMimeType` properties are stored in the database. Listing 12-13 shows the required changes to the `SaveProduct` method.

Listing 12-13. Ensuring That the Image Values Are Stored in the Database in the EFProductRepository.cs File

```
...
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;
            dbEntry.ImageData = product.ImageData;
            dbEntry.ImageMimeType = product.ImageMimeType;
        }
    }
    context.SaveChanges();
}
...
```

Implementing the GetImage Action Method

In Listing 12-11, I added an `img` element whose content was obtained through a `GetImage` action method on the `Product` controller. I am going to implement this action method so that I can display images contained in the database. Listing 12-14 shows the definition of the action method.

Listing 12-14. The GetImage Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List(string category, int page = 1) {
            ProductsListViewModel model = new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.Category == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((page - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                        repository.Products.Count() :
                        repository.Products.Where(e => e.Category == category).Count()
                },
                CurrentCategory = category
            };
            return View(model);
        }

        public FileContentResult GetImage(int productId) {
            Product prod = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            if (prod != null) {
                return File(prod.ImageData, prod.ImageMimeType);
            } else {
                return null;
            }
        }
    }
}
```

This method tries to find a product that matches the ID specified by the parameter. The `FileContentResult` class is used as the result from an action method when you want to return a file to the client browser, and instances are created using the `File` method of the base controller class. I discuss the different types of results you can return from action methods in Chapter 17.

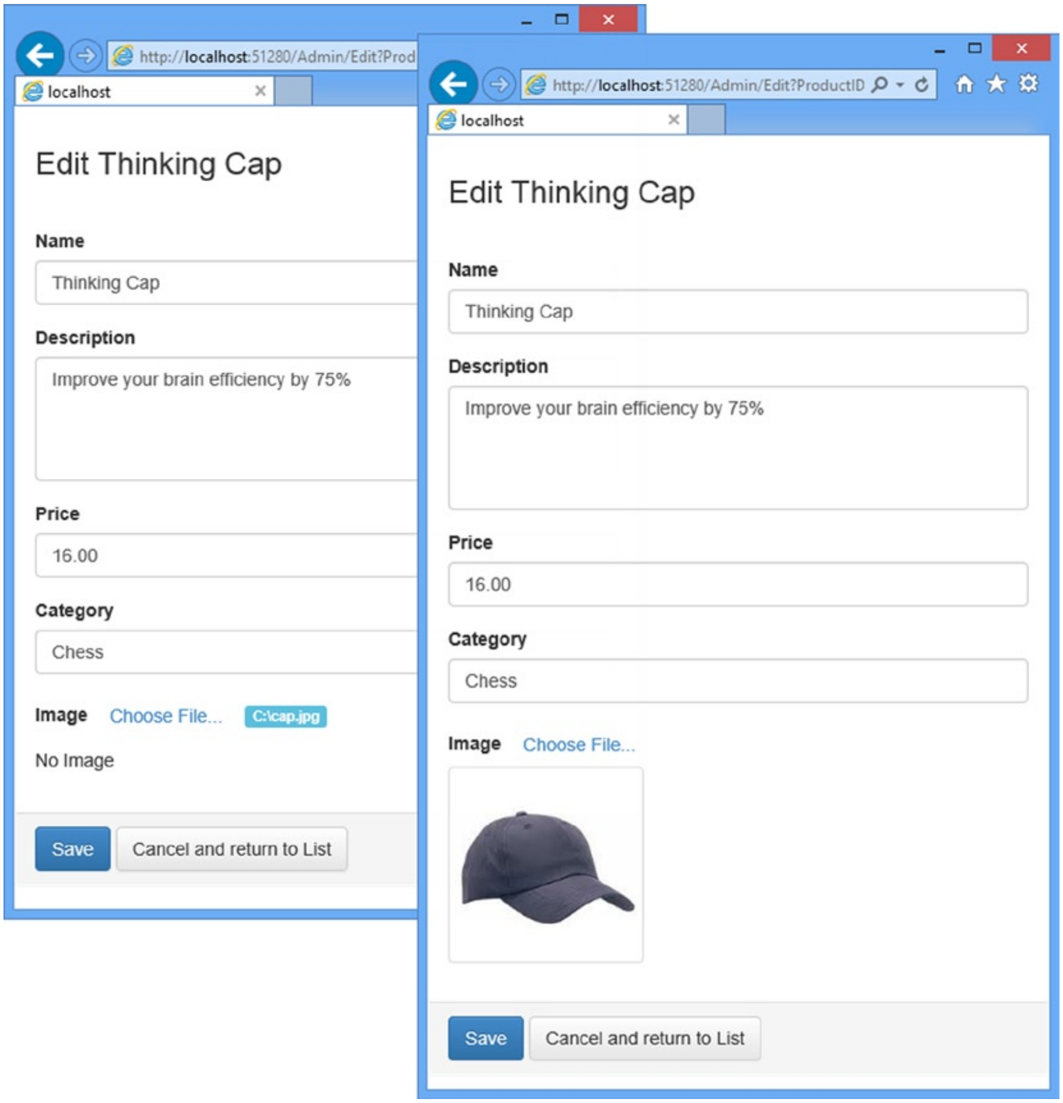
UNIT TEST: RETRIEVING IMAGES

I want to make sure that the `GetImage` method returns the correct MIME type from the repository and make sure that no data is returned when I request a product ID that doesn't exist. Here are the test methods I created, which I defined in a new unit test file called `ImageTests.cs`:

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

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

        [TestMethod]
        public void Can_Retrieve_Image_Data() {

            // Arrange - create a Product with image data
            Product prod = new Product {
                ProductID = 2,
                Name = "Test",
                ImageData = new byte[] { },
                ImageMimeType = "image/png"
            };

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

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

            // Act - call the GetImage action method
            ActionResult result = target.GetImage(2);

            // Assert
            Assert.IsNotNull(result);
            Assert.IsInstanceOfType(result, typeof(FileResult));
            Assert.AreEqual(prod.ImageMimeType, ((FileResult)result).ContentType);
        }
        [TestMethod]
        public void Cannot_Retrieve_Image_Data_For_Invalid_ID() {

            // 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"}
            }.AsQueryable());

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

            // Act - call the GetImage action method
            ActionResult result = target.GetImage(100);

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

When dealing with a valid product ID, I check that I get a `FileResult` result from the action method and that the content type matches the type in the mock data. The `FileResult` class does not let me access the binary contents of the file, so I must be satisfied with a less than perfect test. When I request an invalid product ID, I check to ensure that the result is `null`.

The administrator can now upload images for products. You can try this yourself by starting the application, navigating to the `/Admin/Index` URL and editing one of the products. Figure 12-4 shows an example.


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

Edit Thinking Cap

Name
Thinking Cap

Description
Improve your brain efficiency by 75%

Price
16.00

Category
Chess

Image Choose File... C:\cap.jpg

No Image

Save Cancel and return to List

Edit Thinking Cap

Name
Thinking Cap

Description
Improve your brain efficiency by 75%

Price
16.00

Category
Chess

Image Choose File...

Save Cancel and return to List

Figure 12-4. *Adding an image to a product listing*

Displaying Product Images

All that remains is to display the images alongside the product description in the product catalog. Edit the `Views/Shared/ProductSummary.cshtml` view to reflect the changes shown in bold in Listing 12-15.

Listing 12-15. Displaying Images in the ProductSummary.cshtml File

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

<div class="well">

    @if (Model.ImageData != null) {
        <div class="pull-left" style="margin-right: 10px">
            <img class="img-thumbnail" width="75" height="75"
               src="@Url.Action("GetImage", "Product",
                new { Model.ProductID })" />
        </div>
    }

    <h3>
        <strong>@Model.Name</strong>
        <span class="pull-right label label-primary">@Model.Price.ToString("c")</span>
    </h3>
    @using (Html.BeginForm("AddToCart", "Cart")) {
        <div class="pull-right">
            @Html.HiddenFor(x => x.ProductID)
            @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
            <input type="submit" class="btn btn-success" value="Add to cart" />
        </div>
    }
    <span class="lead"> @Model.Description</span>
</div>
```

With these changes in place, the customers will see images displayed as part of the product description when they browse the catalog, as shown in Figure 12-5.

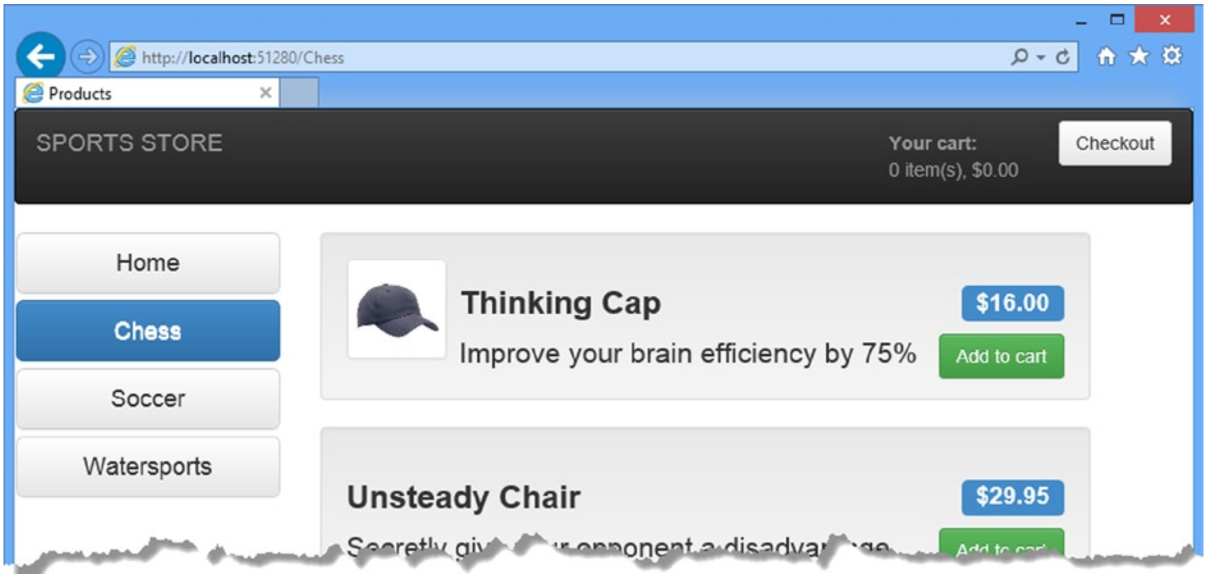


Figure 12-5. *Displaying product images*

Summary

In this and previous chapters, I have demonstrated how the ASP.NET MVC Framework can be used to create a realistic e-commerce application. This extended example has introduced many of the framework's key features: controllers, action methods, routing, views, model binding, metadata, validation, layouts, authentication, and more. You have also seen how some of the key technologies related to MVC can be used. These included the Entity Framework, Ninject, Moq, and the Visual Studio support for unit testing. The result is an application that has a clean, component-oriented architecture that separates out the various concerns, and a code base that will be easy to extend and maintain. In the next chapter, I show you how to deploy the SportsStore application into production.