Adding Catalog Management

The usual software convention for managing collections of items is to present the user with two types of screens: *list* and *edit* (Figure 6-1). Together, these allow a user to create, read, update, and delete items in that collection. (Collectively, these features are known by the acronym *CRUD*.)

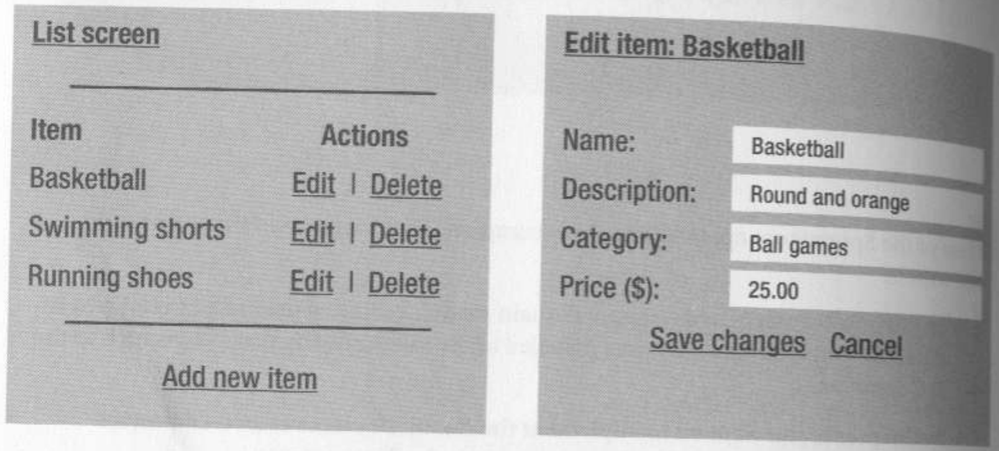


Figure 6-1. *Sketch of a CRUD UI for the product catalog*

CRUD is one of those features that web developers have to implement frequently. So frequently, in fact, that Visual Studio tries to help by offering to automatically generate CRUD-related controllers and view templates for your custom model objects.

Note In this chapter, we'll use Visual Studio's built-in templates occasionally. However, in most cases we'll edit, trim back, or replace entirely the automatically generated CRUD code, because we can make it much more concise and better suited to our task. After all, SportsStore is supposed to be a fairly realistic application, not just demoware specially crafted to make ASP.NET MVC look good.

Creating AdminController: A Place for the CRUD Features

Let's implement a simple CRUD UI for SportsStore's product catalog. Rather than overburdening `ProductsController`, create a new controller class called `AdminController` (right-click the `/Controllers` folder and choose Add ➤ Controller).

Note I made the choice to create a new controller here, rather than simply to extend `ProductsController`, as a matter of personal preference. There's actually no limit to the number of action methods you can put on a single controller. As with all object-oriented programming, you're free to arrange methods and responsibilities any way you like. Of course, it's preferable to keep things organized, so think about the single responsibility principle and break out a new controller when you're switching to a different segment of the application.

If you're interested in seeing the CRUD code that Visual Studio generates, check "Add action methods for Create, Update, and Details scenarios" before clicking Add. It will generate a class that looks like the following:[1]

```
public class AdminController : Controller
{
    public ActionResult Index() { return View(); }

    public ActionResult Details(int id) { return View(); }

    public ActionResult Create() { return View(); }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Create(FormCollection collection)
    {
        try {
            // TODO: Add insert logic here
            return RedirectToAction("Index");
        }
        catch {
            return View();
        }
    }

    public ActionResult Edit(int id) { return View(); }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(int id, FormCollection collection)
    {
        try {
            // TODO: Add update logic here
            return RedirectToAction("Index");
        }
        catch {
            return View();
        }
    }
}
```

The automatically generated code isn't ideal for SportsStore. Why?

It's not yet clear that we're actually going to need all of those methods. Do we really want a Details action? Also, filling in the blanks in automatically generated code may sometimes be a legitimate workflow, but it stands contrary to TDD. Test-first development implies that those action methods shouldn't even exist until we've established, by writing tests, that they are required and should behave in a particular way.

We can write cleaner code than that. We can use model binding to receive edited Product instances as action method parameters. Plus, we definitely don't want to catch and swallow all possible exceptions, as Edit() does by default, as that would ignore and discard important information such as errors thrown by the database when trying to save records.

Don't misunderstand: I'm not saying that using Visual Studio's code generation is always wrong. In fact, the whole system of controller and view code generation can be customized using the powerful T4 templating engine. It's possible to create and share code templates that are ideally suited to your own application's conventions and design guidelines. It could be a fantastic way to get new developers quickly up to speed with your coding practices. However, for now we'll write code manually, because it isn't difficult and it will give you a better understanding of how ASP.NET MVC works.

So, rip out all the automatically generated action methods from AdminController, and then add an IoC dependency on the products repository, as follows:

```
public class AdminController : Controller
{
    private IProductsRepository productsRepository;
    public AdminController(IProductsRepository productsRepository)
    {
        this.productsRepository = productsRepository;
    }
}
```

To support the list screen (shown in Figure 6-1), you'll need to add an action method that displays all products. Following ASP.NET MVC conventions, let's call it Index.

TESTING: THE INDEX ACTION

AdminController's Index action can be pretty simple. All it has to do is render a view, passing all products in the repository. Drive that requirement by adding a new [TestFixture] class, AdminControllerTests, to your Tests project:

```
[TestFixture]
public class AdminControllerTests
{
    // Will share this same repository across all the AdminControllerTests
    private Moq.Mock<IProductsRepository> mockRepos;

    // This method gets called before each test is run
    [SetUp]
```

```
    public void SetUp()
    {
        // Make a new mock repository with 50 products
        List<Product> allProducts = new List<Product>();
        for (int i = 1; i <= 50; i++)
            allProducts.Add(new Product {ProductID = i, Name = "Product " + i});
        mockRepos = new Moq.Mock<IProductsRepository>();
        mockRepos.Setup(x => x.Products)
                 .Returns(allProducts.AsQueryable());
    }

    [Test]
    public void Index_Action_Lists_All_Products()
    {
        // Arrange
        AdminController controller = new AdminController(mockRepos.Object);

        // Act
        ViewResult results = controller.Index();

        // Assert: Renders default view
        Assert.IsEmpty(results.ViewName);
        // Assert: Check that all the products are included
        var prodsRendered = (List<Product>)results.ViewData.Model;
        Assert.AreEqual(50, prodsRendered.Count);
        for (int i = 0; i < 50; i++)
            Assert.AreEqual("Product " + (i + 1), prodsRendered[i].Name);
    }
}
```

This time, we're creating a single shared mock products repository (mockRepos, containing 50 products) to be reused in all the AdminControllerTests tests (unlike CartControllerTests, which constructs a different mock repository tailored to each test case). Again, there's no officially right or wrong technique. I just want to show you a few different approaches so you can pick what appears to work best in each situation.

This test drives the requirement for an Index() action method on AdminController. In other words, there's a compiler error because that method is missing. Let's add it.

Rendering a Grid of Products in the Repository

Add a new action method to AdminController called Index:

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

Trivial as it is, that's enough to make `Index_Action_Lists_All_Products()` pass. You now just need to create a suitable view template that renders those products into a grid, and then the CRUD list screen will be complete.

Implementing the List View Template

Actually, before we add a new view template to act as the view for this action, let's create a new master page for the whole administrative section. In Solution Explorer, right-click the `/Views/Shared` folder, choose Add ➤ New Item, and then from the pop-up window select MVC View Master Page, and call it `Admin.Master`. Put in it the following markup:

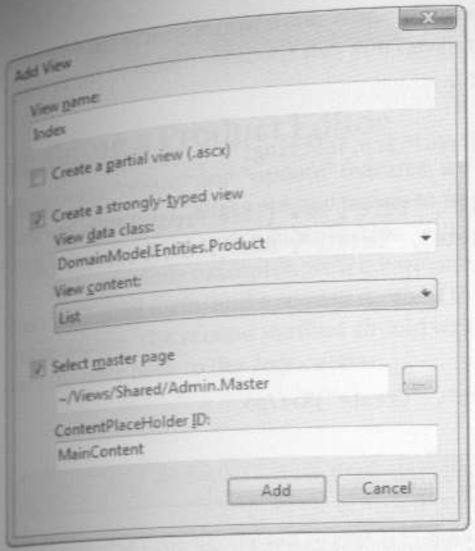
```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
    <head runat="server">
        <link rel="Stylesheet" href="~/Content/adminstyles.css" />
        <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
    </head>
    <body>
        <asp:ContentPlaceHolder ID="MainContent" runat="server" />
    </body>
</html>
```

This master page references a CSS file, so create one called `adminstyles.css` in the `/Content` folder, containing the following:

```
BODY, TD { font-family: Segoe UI, Verdana }
H1 { padding: .5em; padding-top: 0; font-weight: bold;
     font-size: 1.5em; border-bottom: 2px solid gray; }
DIV#content { padding: .9em; }
TABLE.Grid TD, TABLE.Grid TH { border-bottom: 1px dotted gray; text-align:left; }
TABLE.Grid { border-collapse: collapse; width:100%; }
TABLE.Grid TH.NumericCol, Table.Grid TD.NumericCol {
    text-align: right; padding-right: 1em; }
DIV.Message { background: gray; color:White; padding: .2em; margin-top:.25em; }

.field-validation-error { color: red; }
.input-validation-error { border: 1px solid red; background-color: #ffeeee; }
.validation-summary-errors { font-weight: bold; color: red; }
```

Now that you've created the master page, you can add a view template for `AdminController`'s `Index` action. Right-click inside the action method and choose Add View, and then configure the new view template, as shown in Figure 6-2. Notice that the master page is set to `Admin.Master` (not the usual `Site.Master`). Also, on this occasion, we're asking Visual Studio to prepopulate the new view with markup to render a list of `Product` instances.



View name: Index
[ ] Create a partial view (.ascx)
[✓] Create a strongly-typed view
View data class: DomainModel.Entities.Product
View content: List
[✓] Select master page
~/Views/Shared/Admin.Master
ContentPlaceHolder ID: MainContent

Figure 6-2. *Settings for the Index view*

■ Note When you set "View content" to List, Visual Studio implicitly assumes that the view data class should be `IEnumerable<yourclass>`. This means you don't need to type in `IEnumerable<...>` manually.

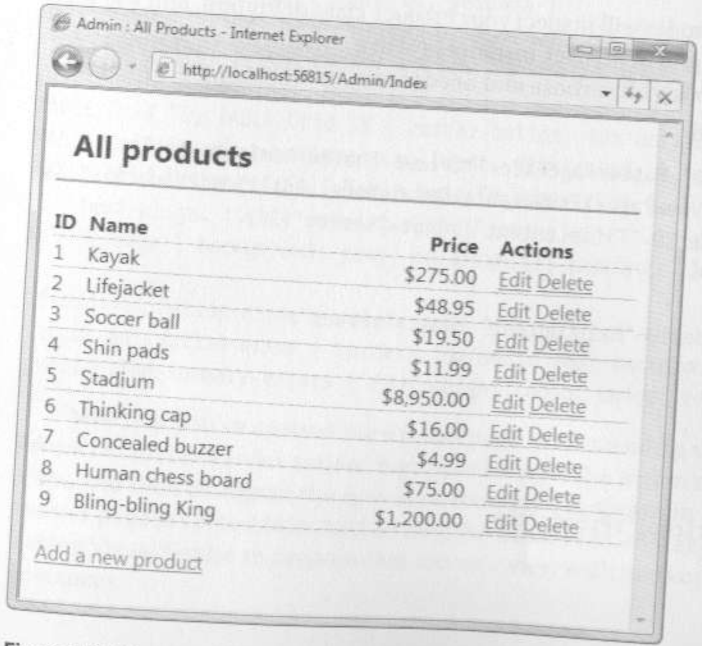
When you click Add, Visual Studio will inspect your `Product` class definition, and will then generate markup for rendering a grid of `Product` instances (with a column for each property on the class). The default markup is a bit verbose and needs some tweaking to match our CSS rules. Edit it to form the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Admin.Master"
   Inherits="System.Web.Mvc.ViewPage<IEnumerable<DomainModel.Entities.Product>>" %>
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
   Admin : All Products
</asp:Content>
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
   <h1>All products</h1>
   <table class="Grid">
      <tr>
         <th>ID</th>
         <th>Name</th>
         <th class="NumericCol">Price</th>
         <th>Actions</th>
      </tr>
```

```
        <% foreach (var item in Model) { %>
            <tr>
                <td><%= item.ProductID %></td>
                <td><%= item.Name %></td>
                <td class="NumericCol"><%= item.Price.ToString("c") %></td>
                <td>
                    <%= Html.ActionLink("Edit", "Edit", new {item.ProductID}) %>
                    <%= Html.ActionLink("Delete", "Delete", new {item.ProductID})%>
                </td>
            </tr>
        <% } %>
    </table>
    <p><%= Html.ActionLink("Add a new product", "Create")%></p>
</asp:Content>
```

Note This view template does not HTML-encode the product details as it renders them. That's fine as long as only administrators are able to edit those details. If, however, you allowed untrusted visitors to submit or edit product information, it would be vital to use `Html.Encode()` to block XSS attacks. See Chapter 13 for more details about this.

You can check this out by launching the application in debug mode (press F5), and then pointing your browser at `http://localhost:port/Admin/Index`, as shown in Figure 6-3.


Admin : All Products - Internet Explorer

http://localhost:56815/Admin/Index

All products

ID	Name	Price	Actions
1	Kayak	$275.00	Edit Delete
2	Lifejacket	$48.95	Edit Delete
3	Soccer ball	$19.50	Edit Delete
4	Shin pads	$11.99	Edit Delete
5	Stadium	$8,950.00	Edit Delete
6	Thinking cap	$16.00	Edit Delete
7	Concealed buzzer	$4.99	Edit Delete
8	Human chess board	$75.00	Edit Delete
9	Bling-bling King	$1,200.00	Edit Delete

Add a new product

Figure 6-3. *The administrator's product list screen*

The list screen is now done. None of its edit/delete/add links work yet, however, because they point to action methods that you haven't yet created. So let's add them next.

Building a Product Editor

To provide "create" and "update" features, we'll now add a product-editing screen along the lines of Figure 6-1. There are two halves to its job: firstly, displaying the edit screen, and secondly, handling the user's submissions.

As in previous examples, we'll create one method that responds to GET requests and renders the initial form, and a second method that responds to POST requests and handles form submissions. The second method should write the incoming data to the repository and redirect the user back to the `Index` action.

TESTING: THE EDIT ACTION

If you're following along in TDD mode, now's the time to add a test for the GET overload of the `Edit` action. You need to verify that, for example, `Edit(17)` renders its default view, passing `Product 17` from the mock products repository as the model object to render. The "assert" phase of the test would include something like this:

```
Product renderedProduct = (Product)result.ViewData.Model;
Assert.AreEqual(17, renderedProduct.ProductID);
Assert.AreEqual("Product 17", renderedProduct.Name);
```

By attempting to call an `Edit()` method on `AdminController`, which doesn't yet exist, this test will cause a compiler error. That drives the requirement to create the `Edit()` method. If you prefer, you could first create a method stub for `Edit()` that simply throws a `NotImplementedException`—that keeps the compiler and IDE happy, leaving you with a red light in NUnit GUI (driving the requirement to implement `Edit()` properly). Whether or not you create such method stubs is another matter of personal preference. I do, because compiler errors make me feel queasy.

The full code for this test is included in the book's downloadable code.

All `Edit()` needs to do is retrieve the requested product and pass it as `Model` to some view. Here's the code you need to add to the `AdminController` class:

```
[AcceptVerbs(HttpVerbs.Get)]
public ViewResult Edit(int productId)
{
    Product product = (from p in productsRepository.Products
                       where p.ProductID == productId
                       select p).First();
    return View(product);
}
```

Creating a Product Editor UI

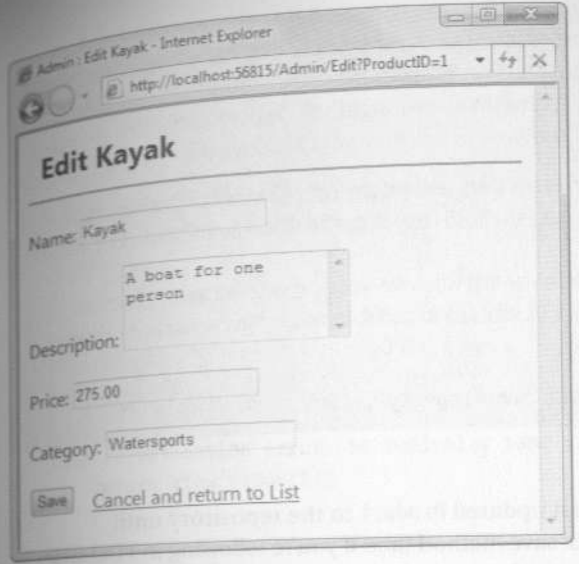
Of course, you'll need to add a view for this. Add a new view template for the Edit action, specifying Admin.Master as its master page, and making it strongly typed for the Product class.

If you like, you can set the "View content" option to Edit, which will cause Visual Studio to generate a basic Product-editing view. However, the resulting markup is again somewhat verbose and much of it is not required. Either set "View content" to Empty, or at least edit the generated markup to form the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Admin.Master"
    Inherits="System.Web.Mvc.ViewPage<DomainModel.Entities.Product>" %>
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
    Admin : Edit <%= Model.Name %>
</asp:Content>
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h1>Edit <%= Model.Name %></h1>

    <% using (Html.BeginForm()) {%>
        <%= Html.Hidden("ProductID") %>
        <p>
            Name: <%= Html.TextBox("Name") %>
            <div><%= Html.ValidationMessage("Name") %></div>
        </p>
        <p>
            Description: <%= Html.TextArea("Description", null, 4, 20, null) %>
            <div><%= Html.ValidationMessage("Description") %></div>
        </p>
        <p>
            Price: <%= Html.TextBox("Price") %>
            <div><%= Html.ValidationMessage("Price") %></div>
        </p>
        <p>
            Category: <%= Html.TextBox("Category") %>
            <div><%= Html.ValidationMessage("Category") %></div>
        </p>
        <input type="submit" value="Save" /> &nbsp;&nbsp;
        <%=Html.ActionLink("Cancel and return to List", "Index") %>
    <% } %>
</asp:Content>
```

It's not the slickest design ever seen, but you can work on the graphics later. You can reach this page by going to /Admin/Index (the "All Products" screen), and then clicking any of the existing edit links. That will bring up the product editor you just created (Figure 6-4).


Admin : Edit Kayak - Internet Explorer
http://localhost:56815/Admin/Edit?ProductID=1

Edit Kayak

Name: Kayak

Description: A boat for one person

Price: 275.00

Category: Watersports

Save   Cancel and return to List

Figure 6-4. *The product editor*

Handling Edit Submissions

If you submit this form, you'll get a 404 Not Found error. That's because there isn't an action method called Edit() that's willing to respond to POST requests. The next job is to add one.

TESTING: EDIT SUBMISSIONS

Before implementing the POST overload of the Edit() action method, add a new test to AdminControllerTests that defines and verifies that action's behavior. You should check that, when passed a Product instance, the method saves it to the repository by calling productsRepository.SaveProduct() (a method that doesn't yet exist). Then it should redirect the visitor back to the Index action.

Here's how you can test all that:

```
[Test]
public void Edit_Action_Saves_Product_To_Repository_And_Redirects_To_Index()
{
    // Arrange
    AdminController controller = new AdminController(mockRepos.Object);
    Product newProduct = new Product();

    // Act
    var result = (RedirectToRouteResult)controller.Edit(newProduct);
```

Continued

```
    // Assert: Saved product to repository and redirected
    mockRepos.Verify(x => x.SaveProduct(newProduct));
    Assert.AreEqual("Index", result.RouteValues["action"]);
}
```

This test will give rise to a few compiler errors: there isn't yet any Edit() overload that accepts a Product instance as a parameter, and IProductsRepository doesn't define a SaveProduct() method. We'll fix that next.

You could also add a test to define the behavior that when the incoming data is invalid, the action method should simply redisplay its default view. To simulate invalid data, add to the // Arrange phase of the test a line similar to the following:

```
controller.ModelState.AddModelError("SomeProperty", "Got invalid data");
```

You can't get very far with saving an updated Product to the repository until IProductsRepository offers some kind of save method (and if you're following in TDD style, your last test will be causing compiler errors for want of a SaveProduct() method). Update IProductsRepository:

```
public interface IProductsRepository
{
    IQueryable<Product> Products { get; }
    void SaveProduct(Product product);
}
```

You'll now get more compiler errors because neither of your two concrete implementations, FakeProductsRepository and SqlProductsRepository, expose a SaveProduct() method. It's always party time with the C# compiler! To FakeProductsRepository, you can simply add a stub that throws a NotImplementedException, but for SqlProductsRepository, add a real implementation:

```
public void SaveProduct(Product product)
{
    // If it's a new product, just attach it to the DataContext
    if (product.ProductID == 0)
        productsTable.InsertOnSubmit(product);
    else {
        // If we're updating an existing product, tell the DataContext
        // to be responsible for saving this instance
        productsTable.Attach(product);
        // Also tell the DataContext to detect any changes since the last save
        productsTable.Context.Refresh(RefreshMode.KeepCurrentValues, product);
    }

    productsTable.Context.SubmitChanges();
}
```

At this point, you're ready to implement a POST-handling overload of the Edit() action method on AdminController. The view template at /Views/Admin/Edit.aspx has input controls with names corresponding to the properties on Product, so when the form posts to an action method, you can use model binding to receive a Product instance as an action method parameter. All you have to do then is save it to the repository. Here goes:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Product product)
{
    if (ModelState.IsValid) {
        productsRepository.SaveProduct(product);
        TempData["message"] = product.Name + " has been saved.";
        return RedirectToAction("Index");
    }
    else // Validation error, so redisplay same view
        return View(product);
}
```

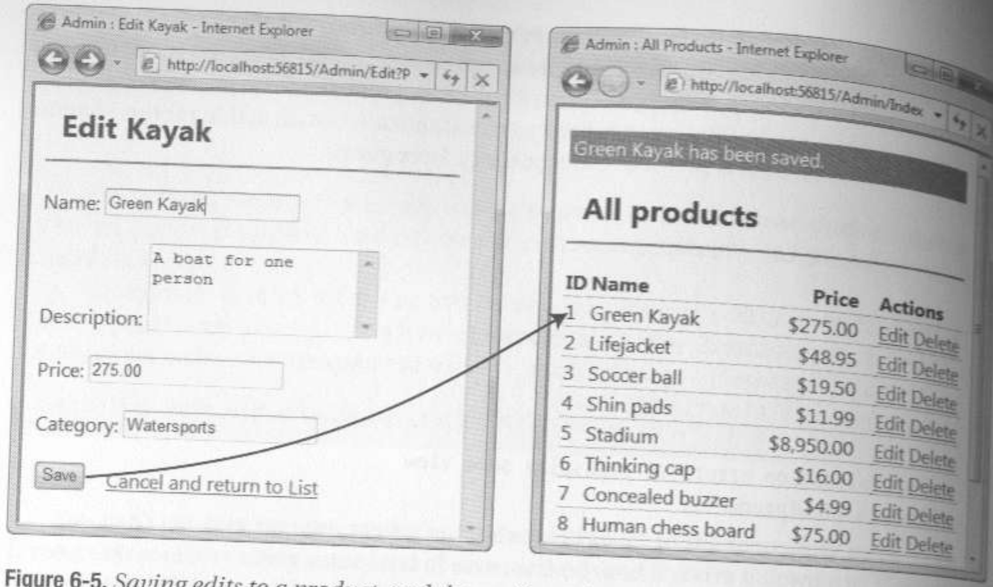
Displaying a Confirmation Message

Notice that after the data gets saved, this action adds a confirmation message to the TempData collection. So, what's TempData? It's a new concept for ASP.NET MVC (traditional WebForms has nothing corresponding to TempData, although other web application platforms do). It's like the Session collection, except that its values survive only for one more HTTP request, and then they're ejected. In this way, TempData tidies up after itself automatically, making it easy to preserve data (e.g., status messages) across HTTP redirections but for no longer.

Since the value in TempData["message"] will be preserved for exactly one further request, you can display it after the HTTP 302 redirection by adding code to the /Views/Shared/Admin.Master master page template:

```
...
<body>
    <% if(TempData["message"] != null) { %>
        <div class="Message"><%= Html.Encode(TempData["message"]) %></div>
    <% } %>
    <asp:ContentPlaceHolder ID="MainContent" runat="server" />
</body>
...
```

Give it a whirl in your browser. You can now update Product records, and get a cute confirmation message each time you do! (See Figure 6-5.)


Admin : Edit Kayak - Internet Explorer

http://localhost:56815/Admin/Edit?P

Edit Kayak

Name: Green Kayak

Description: A boat for one person

Price: 275.00

Category: Watersports

Save | Cancel and return to List

Admin : All Products - Internet Explorer

http://localhost:56815/Admin/Index

Green Kayak has been saved.

All products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Edit Delete
2	Lifejacket	$48.95	Edit Delete
3	Soccer ball	$19.50	Edit Delete
4	Shin pads	$11.99	Edit Delete
5	Stadium	$8,950.00	Edit Delete
6	Thinking cap	$16.00	Edit Delete
7	Concealed buzzer	$4.99	Edit Delete
8	Human chess board	$75.00	Edit Delete

Figure 6-5. *Saving edits to a product, and the confirmation message*

Adding Validation

As always, you'd better not forget about validation. Right now, somebody could come along and put in blank product names or negative prices. The horror! We'll handle that in the same way that we handled validation on `ShippingDetails` in Chapter 5.

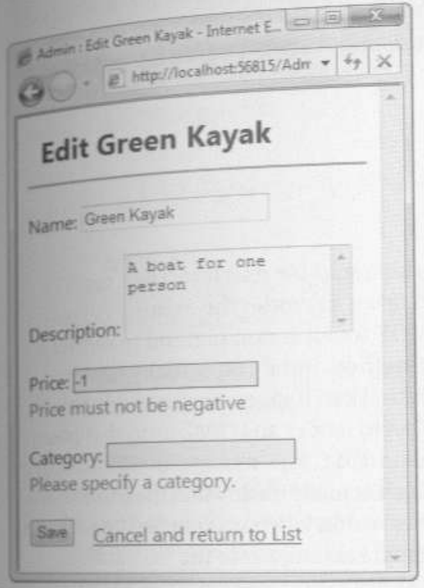
Add code to the `Product` class so that it implements `IDataErrorInfo` as follows:

```
public class Product : IDataErrorInfo
{
    // ... leave everything else as before ...

    public string this[string propName]
    {
        get {
            if ((propName == "Name") && string.IsNullOrEmpty(Name))
                return "Please enter a product name";
            if ((propName == "Description") && string.IsNullOrEmpty(Description))
                return "Please enter a description";
            if ((propName == "Price") && (Price < 0))
                return "Price must not be negative";
            if ((propName == "Category") && string.IsNullOrEmpty(Category))
                return "Please specify a category";
            return null;
        }
    }

    public string Error { get { return null; } } // Not required
}
```

The `IDataErrorInfo` interface will be detected and used by ASP.NET MVC's model binding system. Since the `Edit.aspx` view template renders an `Html.ValidationMessage()` helper for each model property, any validation error messages will be displayed directly next to the invalid control, as shown in Figure 6-6. (This is an alternative to the `Html.ValidationSummary()` helper, which is used to display all the messages in one place.)


Admin : Edit Green Kayak - Internet E...

http://localhost:56815/Adm

Edit Green Kayak

Name: Green Kayak

Description: A boat for one person

Price: -1

Price must not be negative

Category:

Please specify a category.

Save | Cancel and return to List

Figure 6-6. *Validation rules are now enforced, and error messages are displayed next to the offending input controls.*

You might also like to update `SqlProductsRepository` to ensure that it will never save an invalid `Product` to the database, even if some future badly behaving controller asks it to do so. Add to `SqlProductsRepository` an `EnsureValid()` method, and update its `SaveProduct()` method, as follows:

```
public void SaveProduct(Product product)
{
    EnsureValid(product, "Name", "Description", "Category", "Price");

    // ... rest of code as before
}

private void EnsureValid(IDataErrorInfo validatable, params string[] properties)
{
    if (properties.Any(x => validatable[x] != null))
        throw new InvalidOperationException("The object is invalid.");
}
```

Creating New Products

I'm not sure whether you've noticed, but the administrative list screen currently has an "Add a new product" link. It goes to a 404 Not Found error, because it points to an action method called `Create`, which doesn't yet exist.

You need to create a new action method, `Create()`, that deals with adding new `Product` objects. That's easy: all you have to do is render a blank new `Product` object in the existing edit screen. When the user clicks Save, the existing code should save their submission as a new `Product` object. So, to render a blank `Product` into the existing `/Views/Admin/Edit.aspx` view, add the following to `AdminController`:

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

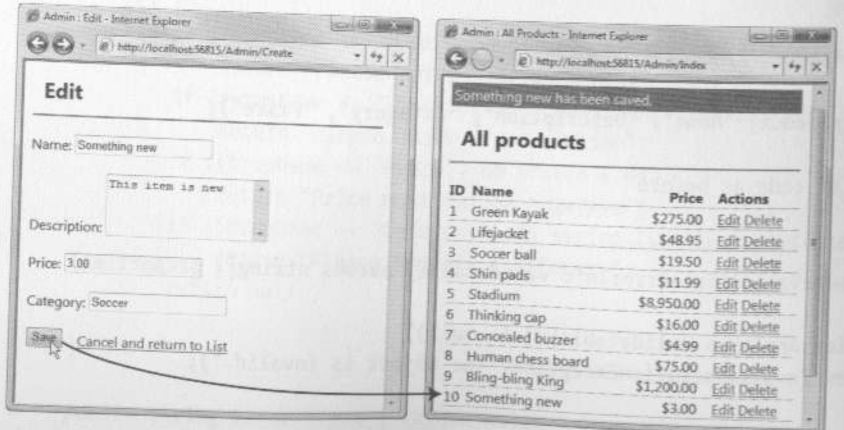
Naturally, you can precede this implementation with a suitable unit test. The `Create()` method does not render its default view, but instead chooses to render the existing `/Views/Admin/Edit.aspx` view. This illustrates that it's perfectly OK for an action method to render a view that's normally associated with a different action method, but if you actually run the application, you'll find that it also illustrates a problem that can happen when you do this.

Typically, you expect the `/Views/Admin/Edit.aspx` view to render an HTML form that posts to the `Edit` action on `AdminController`. However, `/Views/Admin/Edit.aspx` renders its HTML form by calling `Html.BeginForm()` and passing no parameters, which actually means that the form should post to whatever URL the user is currently visiting. In other words, when you render the `Edit` view from the `Create` action, the HTML form will post to the `Create` action, *not* to the `Edit` action.

In this case, we always want the form to post to the `Edit` action, because that's where we've put the logic for saving `Product` instances to the repository. So, edit `/Views/Admin/Edit.aspx`, specifying explicitly that the form should post to the `Edit` action:

```
<% using (Html.BeginForm("Edit", "Admin")) { %>
```

Now the Create functionality will work properly, as shown in Figure 6-7. Validation will happen out of the box, because you've already coded that into the `Edit` action.


Admin : Edit - Internet Explorer
http://localhost:56815/Admin/Create
Admin : All Products - Internet Explorer
http://localhost:56815/Admin/Index

Edit

Name: Something new

Description: This item is new

Price: 3.00

Category: Soccer

Save   Cancel and return to List

Something new has been saved.

All products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Edit Delete
2	Lifejacket	$48.95	Edit Delete
3	Soccer ball	$19.50	Edit Delete
4	Shin pads	$11.99	Edit Delete
5	Stadium	$8,950.00	Edit Delete
6	Thinking cap	$16.00	Edit Delete
7	Concealed buzzer	$4.99	Edit Delete
8	Human chess board	$75.00	Edit Delete
9	Bling-bling King	$1,200.00	Edit Delete
10	Something new	$3.00	Edit Delete

Deleting Products

Deletion is similarly trivial. Your product list screen already renders, for each product, a link to an as-yet-unimplemented action called `Delete`.

TESTING: THE DELETE ACTION

If you're driving this development using tests, you'll need a test that asserts the requirement for a `Delete()` action method. The `Delete()` method should call some kind of delete method on `IProductsRepository`, perhaps along these lines:

```
[Test]
public void Delete_Action_Deletes_Product_Then_Redirects_To_Index()
{
    // Arrange
    AdminController controller = new AdminController(mockRepos.Object);
    Product prod24 = mockRepos.Object.Products.First(p => p.ProductID == 24);

    // Act (attempt to delete product 24)
    RedirectToRouteResult result = controller.Delete(24);

    // Assert
    Assert.AreEqual("Index", result.RouteValues["action"]);
    Assert.AreEqual("Product 24 has been deleted",
                    controller.TempData["message"]);
    mockRepos.Verify(x => x.DeleteProduct(prod24));
}
```

Notice how it uses Moq's `.Verify()` method to ensure that `AdminController` really did call `DeleteProduct()` with the correct parameter. Also, it checks that a suitable notification gets stored in `TempData["message"]` (remember that `/Views/Shared/Admin.Master` is already coded up to display any such message).

To get this working, you'll first need to add a delete method to `IProductsRepository`:

```
public interface IProductsRepository
{
    IQueryable<Product> Products { get; }
    void SaveProduct(Product product);
    void DeleteProduct(Product product);
}
```

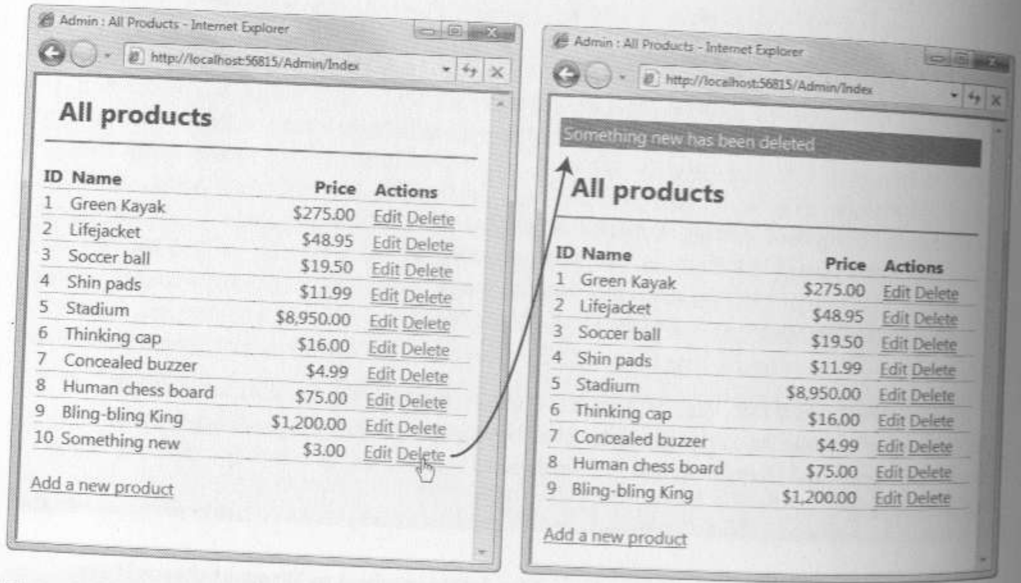
Here's an implementation for `SqlProductsRepository` (you can just throw a `NotImplementedException` in `FakeProductsRepository`):

```
public void DeleteProduct(Product product)
{
    productsTable.DeleteOnSubmit(product);
```

You've already created Delete links from the product list screen. All that's needed now is to actually create an action method called `Delete()`.

Get the test to compile and pass by implementing a real `Delete()` action method on `AdminController` as follows. This results in the functionality shown in Figure 6-8.

```
public RedirectToRouteResult Delete(int productId)
{
    Product product = (from p in productsRepository.Products
                       where p.ProductID == productId
                       select p).First();
    productsRepository.DeleteProduct(product);
    TempData["message"] = product.Name + " has been deleted";
    return RedirectToAction("Index");
}
```


Admin : All Products - Internet Explorer

http://localhost:56815/Admin/Index

All products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Edit Delete
2	Lifejacket	$48.95	Edit Delete
3	Soccer ball	$19.50	Edit Delete
4	Shin pads	$11.99	Edit Delete
5	Stadium	$8,950.00	Edit Delete
6	Thinking cap	$16.00	Edit Delete
7	Concealed buzzer	$4.99	Edit Delete
8	Human chess board	$75.00	Edit Delete
9	Bling-bling King	$1,200.00	Edit Delete
10	Something new	$3.00	Edit Delete

Add a new product

Admin : All Products - Internet Explorer

http://localhost:56815/Admin/Index

Something new has been deleted

All products

ID	Name	Price	Actions
1	Green Kayak	$275.00	Edit Delete
2	Lifejacket	$48.95	Edit Delete
3	Soccer ball	$19.50	Edit Delete
4	Shin pads	$11.99	Edit Delete
5	Stadium	$8,950.00	Edit Delete
6	Thinking cap	$16.00	Edit Delete
7	Concealed buzzer	$4.99	Edit Delete
8	Human chess board	$75.00	Edit Delete
9	Bling-bling King	$1,200.00	Edit Delete

Add a new product

Figure 6-8. *Deleting a product*

And that's it for catalog management CRUD: you can now create, read, update, and delete `Product` records.

Securing the Administration Features

Hopefully it hasn't escaped your attention that, if you deployed this application right now, anybody could visit `http://`*`yourserver`*`/Admin/Index` and play havoc with your product catalog. You need to stop this by password-protecting the entire `AdminController`.

Setting Up Forms Authentication

ASP.NET MVC is built on the core ASP.NET platform, so you automatically have access to ASP.NET's Forms Authentication facility, which is a general purpose system for keeping track of who's logged in. It can be connected to a range of login UIs and credential stores, including custom ones. You'll learn about Forms Authentication in more detail in Chapter 15, but for now, let's set it up in a simple way.

Open up your `web.config` file and find the `<authentication>` node:

```
<authentication mode="Forms">
    <forms loginUrl="~/Account/LogOn" timeout="2880"/>
</authentication>
```

As you can see, brand-new ASP.NET MVC applications are already set up to use Forms Authentication by default. The `loginUrl` setting tells Forms Authentication that, when it's time for a visitor to log in, it should redirect them to `/Account/LogOn` (which should produce an appropriate login page).

■ **Note** The other main authentication mode is Windows, which means that the web server (IIS) is responsible for determining each HTTP request's security context. That's great if you're building an intranet application in which the server and all client machines are part of the same Windows domain. Your application will be able to recognize visitors by their Windows domain logins and Active Directory roles.

However, Windows Authentication isn't so great for applications hosted on the public Internet, because no such security context exists there. That's why you have another option, Forms, which relies on you providing some other means of authentication (e.g., your own database of login names and passwords). Then Forms Authentication remembers that the visitor is logged in by using browser cookies. That's basically what you want for SportsStore.

The default ASP.NET MVC project template gives you a suggested implementation of `AccountController` and its `LogOn` action (by default, accessible at `/Account/LogOn`), which uses the core ASP.NET membership facility to manage user accounts and passwords. You'll learn more about membership and how you can use it with ASP.NET MVC in Chapter 15. For this chapter's application, however, such a heavyweight system is overkill. In fact, in Chapter 4, you already deleted the initial `AccountController` from your project. You're about to replace it with a simpler alternative. Update the `<authentication>` node in your `web.config` file:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/LogOn" timeout="2880">
    <credentials passwordFormat="SHA1">
      <user name="admin" password="e9fe51f94eadabf54dbf2fbbd57188b9abee436e" />
    </credentials>
  </forms>
</authentication>
```

Although most applications using Forms Authentication store credentials in a database, here you're keeping things very simple by configuring a hard-coded list of usernames and passwords. Presently, this credentials list includes only one login name, admin, with password mysecret (e9fe51f... is the SHA1 hash of mysecret).

Tip Is there any benefit in storing a hashed password rather than a plain-text one? Yes, a little. It makes it harder for someone who reads your web.config file to use any login credentials they find (they'd have to invert the hash, which is hard or impossible depending on the strength of the password you've hashed). If you're not worried about someone reading your web.config file (e.g., because you don't think anyone else has access to your server), you can configure passwords in plain-text by setting passwordFormat="Clear". Of course, in most applications, this is irrelevant because you won't store credentials in web.config at all; credentials will usually be stored (suitably hashed and salted) in a database. See Chapter 15 for more details.

Using a Filter to Enforce Authentication

So far, so good—you've configured Forms Authentication, but as yet it doesn't make any difference. The application still doesn't require anyone to log in. You *could* enforce authentication by putting code like this at the top of each action method you want to secure:

```
if (!Request.IsAuthenticated)
    FormsAuthentication.RedirectToLoginPage();
```

That would work, but it gets tiresome to keep sprinkling these same two lines of code onto every administrative action method you write. And what if you forget one?

ASP.NET MVC has a useful facility called *filters*. These are .NET attributes that you can "tag" onto any action method or controller, plugging some extra logic into the request handling pipeline. There are different types of filters—action filters, error handling filters, authorization filters—that run at different stages in the pipeline, and the framework ships with default implementations of each type. You'll learn more about using each type of filter, and creating your own custom ones, in Chapter 9.

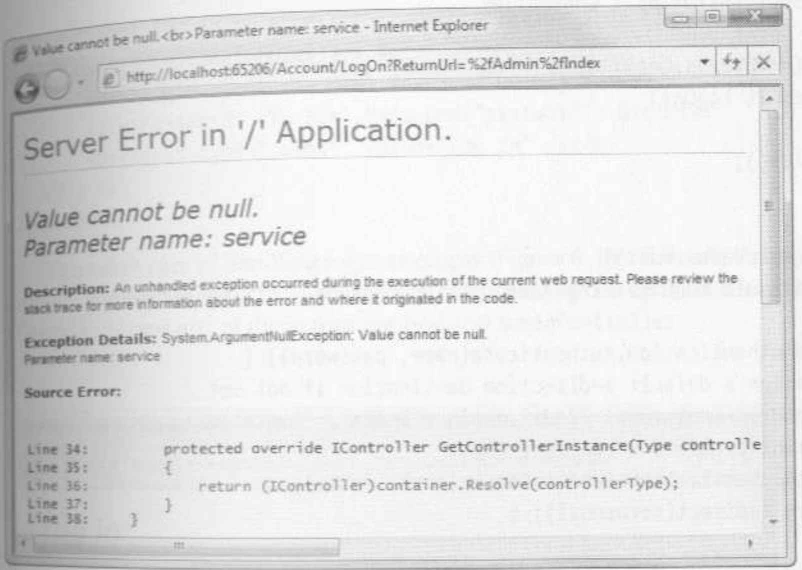
Right now, you can use the default authorization filter,[2] [Authorize]. Simply decorate the AdminController class with [Authorize]:

```
[Authorize]
public class AdminController : Controller
{
    // ... etc
}
```

2. Remember that *authentication* means "identifying a user," while *authorization* means "deciding what a named user is allowed to do." In this simple example, we're treating them as a single concept, saying that a visitor is *authorized* to use AdminController as long as they're *authenticated* (i.e., logged in).

Tip You can attach filters to individual action methods, but attaching them to the controller itself (as in this example) makes them apply to *all* action methods on that controller.

So, what effect does this have? Try it out. If you try to visit /Admin/Index now (or access any action method on AdminController), you'll get the error shown in Figure 6-9.



```
Value cannot be null.<br>Parameter name: service - Internet Explorer
http://localhost:65206/Account/LogOn?ReturnUrl=%2fAdmin%2fIndex

Server Error in '/' Application.

Value cannot be null.
Parameter name: service

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.ArgumentNullException: Value cannot be null.
Parameter name: service

Source Error:

        protected override IController GetControllerInstance(Type controlle
        {
            return (IController)container.Resolve(controllerType);
        }
    }
```

Figure 6-9. *An unauthenticated visitor gets redirected to /Account/LogOn.*

Notice the address bar. It reads as follows:

```
/Account/LogOn?ReturnUrl=%2fAdmin%2fIndex
```

This shows that Forms Authentication has kicked in and redirected the visitor to the URL you configured in web.config (helpfully keeping a record of the original URL they requested in a query string parameter called ReturnUrl). However, there's isn't yet any registered controller class to handle requests for that URL, so your WindsorControllerFactory raises an error.

Displaying a Login Prompt

Your next step is to handle these requests for /Account/LogOn, by adding a controller called AccountController with an action called LogOn.

- There will be a method called LogOn() that handles GET requests. This will render a view for a login prompt.
- There will be another overload of LogOn() that handles POST requests. This overload will ask

- If the credentials are valid, it will tell Forms Authentication to consider the visitor logged in, and will redirect the visitor back to whatever URL originally triggered the [Authorize] filter.
- If the credentials are invalid, it will simply redisplay the login prompt (with a suitable notice saying "Try again").

To achieve all this, create a new controller called AccountController, adding the following action methods:

```
public class AccountController : Controller
{
    [AcceptVerbs(HttpVerbs.Get)]
    public ViewResult LogOn()
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult LogOn(string name, string password, string returnUrl)
    {
        if (FormsAuthentication.Authenticate(name, password)) {
            // Assign a default redirection destination if not set
            returnUrl = returnUrl ?? Url.Action("Index", "Admin");
            // Grant cookie and redirect
            FormsAuthentication.SetAuthCookie(name, false);
            return Redirect(returnUrl); ;
        }
        else {
            ViewData["lastLoginFailed"] = true;
            return View();
        }
    }
}
```

You'll also need a suitable view template for these LogOn() action methods. Add one by right-clicking inside one of the LogOn() methods and choosing Add View. You can uncheck "Create a strongly typed view" because you don't need a strong concept of a model for this simple view. For "Master page," specify ~/Views/Shared/Admin.Master.

Here's the markup needed to render a simple login form:

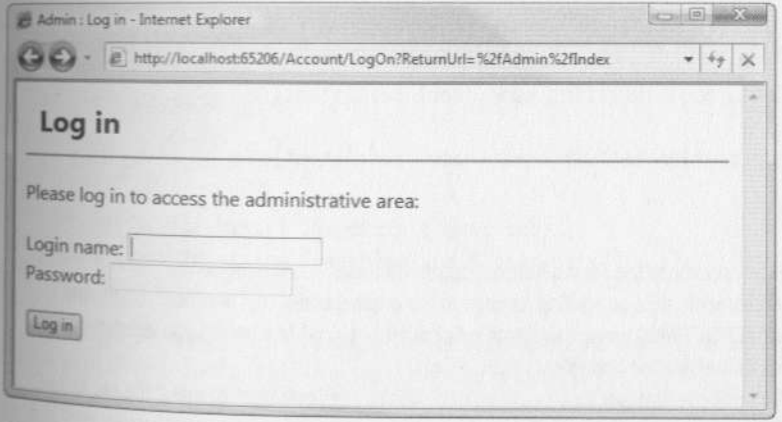
```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Admin.Master"
    Inherits="System.Web.Mvc.ViewPage" %>
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
    Admin : Log in
</asp:Content>
```

```
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h1>Log in</h1>

    <% if((bool?)ViewData["lastLoginFailed"] == true) { %>
        <div class="Message">
            Sorry, your login attempt failed. Please try again.
        </div>
    <% } %>

    <p>Please log in to access the administrative area:</p>
    <% using(Html.BeginForm()) { %>
        <div>Login name: <%= Html.TextBox("name") %></div>
        <div>Password: <%= Html.Password("password") %></div>
        <p><input type="submit" value="Log in" /></p>
    <% } %>
</asp:Content>
```

This takes care of handling login attempts (Figure 6-10). Only after supplying valid credentials (i.e., admin/mysecret) will visitors be granted an authentication cookie and thus be allowed to access any of the action methods on AdminController.


Admin : Log in - Internet Explorer

http://localhost:65206/Account/LogOn?ReturnUrl=%2fAdmin%2fIndex

Log in

Please log in to access the administrative area:

Login name:

Password:

Figure 6-10. *The login prompt (rendered using /Views/Account/LogOn.aspx)*

Caution When you're sending login details from browser to server, it's best to encrypt the transmission with SSL (i.e., over HTTPS). To do this, you need to set up SSL on your web server, which is beyond the scope of this chapter—Visual Studio's built-in web server doesn't support it. See IIS documentation for details about how to configure SSL.

BUT WHAT ABOUT TESTABILITY?

If you're trying to write unit tests for `Login()`, you'll hit a problem. Right now, that code is directly coupled to two static methods on the `FormsAuthentication` class (`Authenticate()` and `SetAuthCookie()`).

Ideally, your unit tests would supply some kind of mock `FormsAuthentication` object, and then they could test `Login()`'s interaction with Forms Authentication (i.e., checking that it calls `SetAuthCookie()` only when `Authenticate()` returns `true`). However, Forms Authentication's API is built around static methods, so there's no easy way to mock it. Forms Authentication is quite an old piece of code, and unlike the modern MVC Framework, it simply wasn't designed with testability in mind.

The normal way to make untestable code testable is to wrap it inside an interface type. You create a class that implements the interface by simply delegating all calls to the original code. For example, add the following types anywhere in your `WebUI` project:

```
public interface IFormsAuth
{
    bool Authenticate(string name, string password);
    void SetAuthCookie(string name, bool persistent);
}
public class FormsAuthWrapper : IFormsAuth
{
    public bool Authenticate(string name, string password)
    {
        return FormsAuthentication.Authenticate(name, password);
    }
    public void SetAuthCookie(string name, bool persistent)
    {
        FormsAuthentication.SetAuthCookie(name, persistent);
    }
}
```

Here, `IFormsAuth` represents the Forms Authentication methods you'll need to call. `FormsAuthWrapper` implements this, delegating its calls to the original code. This is almost exactly the same as how the default ASP.NET MVC project template's `AccountController` (which you deleted in Chapter 4) makes Forms Authentication testable.

In fact, it's also the same mechanism that `System.Web.Abstractions` uses to make the old ASP.NET context classes (e.g., `HttpRequest`) testable, defining abstract base classes (e.g., `HttpRequestBase`) and subclasses (e.g., `HttpRequestWrapper`) that simply delegate to the original code. Microsoft chose to use abstract base classes (with stub implementations of each method) instead of interfaces so that, when subclassing them, you only have to override the specific methods that interest you (whereas with an interface, you must implement all its methods).

Now, there are two main ways of supplying an `IFormsAuth` instance to the `Login()` method:

Using your IoC container. You could register `IFormsAuth` as an IoC component (with `FormsAuthWrapper` configured as its active concrete type) and then have `AccountController` demand an `IFormsAuth` instance as a constructor parameter. At runtime, `WindsorControllerFactory` would take care of supplying a `FormsAuthWrapper` instance. In your tests, you could supply a mock `IFormsAuth` instance as a constructor parameter to `AccountController`.

Using a custom model binder. You could create a custom model binder for `IFormsAuth` that simply returns an instance of `FormsAuthWrapper`. Once your custom model binder is registered (just as you registered `CartModelBinder` in Chapter 5), any of your action methods can demand an `IFormsAuth` object as a method parameter. At runtime, your custom model binder would supply a `FormsAuthWrapper` instance. In your tests, you could supply a mock `IFormsAuth` instance as an action method parameter.

Both approaches are equally good. If you're making heavy use of an IoC container, then you might prefer the first option (which has the benefit that you could swap out `FormsAuthWrapper` for a different authentication mechanism without even recompiling your code). Otherwise, the custom model binder approach is convenient enough.

Image Uploads

Let's finish the whole SportsStore application by implementing something slightly more sophisticated: the ability for administrators to upload product images, store them in the database, and display them on product list screens.

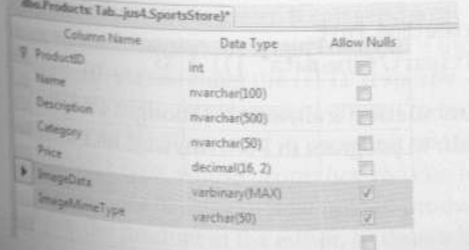
Preparing the Domain Model and Database

To get started, add two extra fields to the `Product` class, which will hold the image's binary data and its MIME type (to specify whether it's a JPEG, GIF, PNG, or other type of file):

```
[Table(Name = "Products")]
public class Product
{
    // Rest of class unchanged

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

Next, use Server Explorer (or SQL Server Management Studio) to add corresponding columns to the `Products` table in your database (Figure 6-11).


dbo.Products: Tab...jus4.SportsStore)*

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

Figure 6-11. *Adding new columns using Server Explorer*

Accepting File Uploads

Next, add a file upload box to /Views/Admin/Edit.aspx:

```
<p>
    Category: <%= Html.TextBox("Category") %>
    <div><%= Html.ValidationMessage("Category") %></div>
</p>
<p>
    Image:
    <% if(Model.ImageData == null) { %>
        None
    <% } else { %>
        <img src="<%= Url.Action("GetImage", "Products",
                                 new { Model.ProductID }) %>" />
    <% } %>
    <div>Upload new image: <input type="file" name="Image" /></div>
</p>

<input type="submit" value="Save" />
<!-- ... rest unchanged ... -->
```

Notice that if the Product being displayed already has a non-null value for ImageData, the view attempts to display that image by rendering an `<img>` tag referencing a not-yet-implemented action on ProductsController called GetImage. We'll come back to that in a moment.

A Little-Known Fact About HTML Forms

In case you weren't aware, web browsers will only upload files properly when the `<form>` tag defines an enctype value of multipart/form-data. In other words, for a successful upload, the `<form>` tag must look like this:

```
<form enctype="multipart/form-data">...</form>
```

Without that enctype attribute, the browser will transmit only the *name* of the file—not its contents—which is no use to us! Force the enctype attribute to appear by updating Edit.aspx's call to Html.BeginForm():

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

Ugh—the end of that line is now a bit of a punctuation trainwreck! I thought I'd left that sort of thing behind when I vowed never again to program in Perl. Anyway, let's move swiftly on.

Saving the Uploaded Image to the Database

OK, so your domain model can store images, and you've got a view that can upload them, so you now need to update AdminController's POST-handling Edit() action method to receive and store that uploaded image data. That's pretty easy: just accept the upload as an HttpPostedFileBase method parameter, and copy its data to the product object:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Product product, HttpPostedFileBase image)
{
    if (ModelState.IsValid) {
        if (image != null) {
            product.ImageMimeType = image.ContentType;
            product.ImageData = new byte[image.ContentLength];
            image.InputStream.Read(product.ImageData, 0, image.ContentLength);
        }
        productsRepository.SaveProduct(product);
        ...
```

Of course, you'll need to update the unit test that calls Edit()—if you have one—to supply some value (such as null) for the image parameter; otherwise, you'll get a compiler error.

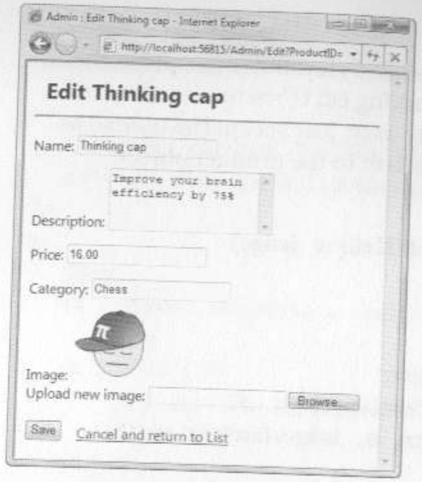
Displaying Product Images

You've implemented everything needed to accept image uploads and store them in the database, but you still don't have the GetImage action that's expected to return image data for display. Add this to ProductsController:

```
public FileContentResult GetImage(int ProductID)
{
    Product product = (from p in productsRepository.Products
                       where p.ProductID == ProductID
                       select p).First();
    return File(product.ImageData, product.ImageMimeType);
}
```

This action method demonstrates the File() method, which lets you return binary content directly to the browser. It can send a raw byte array (as we're doing here to send the image data to the browser), or it can transmit a file from disk, or it can spool the contents of a System.IO.Stream along the HTTP response. The File() method is testable, too: rather than directly accessing the response stream to transmit the binary data (which would force you to simulate an HTTP context in your unit tests), it actually just returns some subclass of the FileResult type, whose properties you can inspect in a unit test.

That does it! You can now upload product images, and they'll be displayed when you reopen the product in the editor, as shown in Figure 6-12.


Admin : Edit Thinking cap - Internet Explorer
http://localhost:56815/Admin/Edit?ProductID=
Browse...
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:

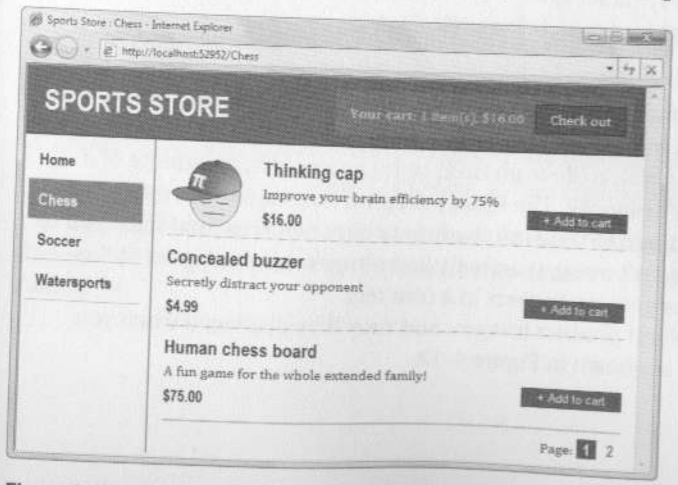
Upload new image:

Figure 6-12. *The product editor, after uploading and saving a product image*

Of course, the real goal is to display product images to the public, so update /Views/Shared/ProductSummary.ascx:

```
<div class="item">
    <% if(Model.ImageData != null) { %>
        <div style="float:left; margin-right:20px">
            <img src="<%= Url.Action("GetImage", "Products",
                          new { Model.ProductID }) %>" />
        </div>
    <% } %>
    <h3><%= Model.Name %></h3>
    ... rest unchanged ...
</div>
```

As Figure 6-13 shows, sales are now likely to increase significantly.


Sports Store : Chess - Internet Explorer
http://localhost:52952/Chess
SPORTS STORE
Your cart: 1 item(s) $16.00
Check out
Home
Chess
Soccer
Watersports
+ Add to cart
+ Add to cart
+ Add to cart
Page: 1 2

Thinking cap
Improve your brain efficiency by 75%
$16.00

Concealed buzzer
Secretly distract your opponent
$4.99

Human chess board
A fun game for the whole extended family!
$75.00

Exercise: RSS Feed of Products

If you'd like to add a final enhancement to SportsStore, consider adding RSS notifications of new products added to the catalog. This will involve the following:

- Adding a new field, CreatedDate, to Product, and the corresponding database column and LINQ to SQL mapping attribute. You can set its value to DateTime.Now when saving a new product.
- Creating a new controller, RssController, perhaps with an action called Feed, that queries the product repository for, say, the 20 most recently added products (in reverse-chronological order), and renders the results as RSS.
- Updating the public master page, /Views/Shared/Site.Master, to notify browsers of the RSS feed by adding a reference to the <head> section—for example:

```
<link rel="alternate" type="application/rss+xml"
title="New SportsStore products" href="http://yourserver/rss/feed" />
```

For reference, here's the kind of output you're aiming for:

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0">
  <channel>
    <title>SportsStore new products</title>
    <description>Buy all the hottest new sports gear</description>
    <link>http://sportsstore.example.com/</link>

    <item>
      <title>Tennis racquet</title>
      <description>Ideal for hitting tennis balls</description>
      <link>http://example.com/tennis</link>
    </item>

    <item>
      <title>Laser-guided bowling ball</title>
      <description>A guaranteed strike, every time</description>
      <link>http://example.com/tenpinbowling</link>
    </item>

  </channel>
</rss>
```

In Chapter 9, you can find an example of an action method using .NET's XDocument API to create RSS data.

Summary

You've now seen how ASP.NET MVC can be used to create a realistic e-commerce application. This extended example demonstrated many of the framework's features (contro