



# SportsStore: Completing the Cart

In this chapter, I continue to build out the SportsStore example app. In the previous chapter, I added the basic support for a shopping cart and now I am going to improve on and complete that functionality.

## Using Model Binding

The MVC Framework uses a system called *model binding* to create C# objects from HTTP requests in order to pass them as parameter values to action methods. This is how the MVC framework processes forms, for example: it looks at the parameters of the action method that has been targeted and uses a *model binder* to get the form values sent by the browser and convert them to the type of the parameter with the same name before passing them to the action method.

Model binders can create C# types from any information that is available in the request. This is one of the central features of the MVC Framework. I am going to create a custom model binder to improve the CartController class.

I like using the session state feature in the Cart controller to store and manage the Cart objects that I set up in Chapter 8, but I do not like the way I have to go about it. It does not fit the rest of the application model, which is based around action method parameters. I cannot properly unit test the CartController class unless I mock the Session parameter of the base class, and that means mocking a whole bunch of other stuff I would rather not deal with.

To solve this problem, I am going to create a custom model binder that obtains the Cart object contained in the session data. The MVC Framework will then be able to create Cart objects and pass them as parameters to the action methods in the CartController class. The model binding feature is powerful and flexible. I go into a lot more depth about this feature in Chapter 24, but this is a nice example to get started with.

## Creating a Custom Model Binder

I create a custom model binder by implementing the `System.Web.Mvc.IModelBinder` interface. To create this implementation, I added a new folder in the SportsStore.WebUI project called Infrastructure/Binders and created a `CartModelBinder.cs` class file inside it. Listing 9-1 shows the contents of the new file.

**Listing 9-1.** The Contents of the CartModelBinder.cs File

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

namespace SportsStore.WebUI.Infrastructure.Binders {

    public class CartModelBinder : IModelBinder {
        private const string sessionKey = "Cart";
```

```

public object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext) {

    // get the Cart from the session

    Cart cart = null;
    if (controllerContext.HttpContext.Session != null) {
        cart = (Cart)controllerContext.HttpContext.Session[sessionKey];
    }
    // create the Cart if there wasn't one in the session data
    if (cart == null) {
        cart = new Cart();
        if (controllerContext.HttpContext.Session != null) {
            controllerContext.HttpContext.Session[sessionKey] = cart;
        }
    }
    // return the cart
    return cart;
}
}
}

```

The `IModelBinder` interface defines one method: `BindModel`. The two parameters are provided to make creating the domain model object possible. The `ControllerContext` provides access to all the information that the controller class has, which includes details of the request from the client. The `ModelBindingContext` gives you information about the model object you are being asked to build and some tools for making the binding process easier.

For my purposes, the `ControllerContext` class is the one I am interested in. It has an `HttpContext` property, which in turn has a `Session` property that lets me get and set session data. I can obtain the `Cart` object associated with the user's session by reading a value from the session data, and create a `Cart` if there is not one there already.

I need to tell the MVC Framework that it can use the `CartModelBinder` class to create instances of `Cart`. I do this in the `Application_Start` method of `Global.asax`, as shown in Listing 9-2.

**Listing 9-2.** Registering the `CartModelBinder` Class in the `Global.asax.cs` File

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Infrastructure.Binders;

namespace SportsStore.WebUI {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}

```

```

        ModelBinders.Binders.Add(typeof(Cart), new CartModelBinder());
    }
}

```

I can now update the Cart controller to remove the GetCart method and rely on the model binder to provide the controller with Cart objects. Listing 9-3 shows the changes.

**Listing 9-3.** Relying on the Model Binder in the CartController.cs File

```

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

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

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

        public ViewResult Index(Cart cart, string returnUrl) {
            return View(new CartIndexViewModel {
                ReturnUrl = returnUrl,
                Cart = cart
            });
        }

        public RedirectToRouteResult AddToCart(Cart cart, int productId,
            string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                cart.AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

        public RedirectToRouteResult RemoveFromCart(Cart cart, int productId,
            string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                cart.RemoveLine(product);
            }
        }
    }
}

```

```

        return RedirectToAction("Index", new { returnUrl });
    }
}
}

```

I have removed the `GetCart` method and added a `Cart` parameter to each of the action methods. When the MVC Framework receives a request that requires, say, the `AddToCart` method to be invoked, it begins by looking at the parameters for the action method. It looks at the list of binders available and tries to find one that can create instances of each parameter type. The custom binder is asked to create a `Cart` object, and it does so by working with the session state feature. Between the custom binder and the default binder, the MVC Framework is able to create the set of parameters required to call the action method, allowing me to refactor the controller so that it has no knowledge of how `Cart` objects are created when requests are received.

There are several benefits to using a custom model binder like this. The first is that I have separated the logic used to create a `Cart` from that of the controller, which allows me to change the way I store `Cart` objects without needing to change the controller. The second benefit is that any controller class that works with `Cart` objects can simply declare them as action method parameters and take advantage of the custom model binder. The third benefit, and the one I think is most important, is that I can now unit test the `Cart` controller without needing to mock a lot of ASP.NET plumbing.

## UNIT TEST: THE CART CONTROLLER

I can unit test the `CartController` class by creating `Cart` objects and passing them to the action methods. I want to test three different aspects of this controller:

- The `AddToCart` method should add the selected product to the customer's cart.
- After adding a product to the cart, the user should be redirected to the `Index` view.
- The URL that the user can follow to return to the catalog should be correctly passed to the `Index` action method.

Here are the unit tests I added to the `CartTests.cs` file in the `SportsStore.UnitTests` project:

```

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

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

        //...existing test methods omitted for brevity...

        [TestMethod]
        public void Can_Add_To_Cart() {

```

```

// 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", Category = "Apples"},
}).AsQueryable();

// Arrange - create a Cart
Cart cart = new Cart();

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

// Act - add a product to the cart
target.AddToCart(cart, 1, null);

// Assert
Assert.AreEqual(cart.Lines.Count(), 1);
Assert.AreEqual(cart.Lines.ToArray()[0].Product.ProductID, 1);
}

[TestMethod]
public void Adding_Product_To_Cart_Goes_To_Cart_Screen() {
    // 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", Category = "Apples"},
    }).AsQueryable();

    // Arrange - create a Cart
    Cart cart = new Cart();

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

    // Act - add a product to the cart
    RedirectToRouteResult result = target.AddToCart(cart, 2, "myUrl");

    // Assert
    Assert.AreEqual(result.RouteValues["action"], "Index");
    Assert.AreEqual(result.RouteValues["returnUrl"], "myUrl");
}

[TestMethod]
public void Can_View_Cart_Contents() {
    // Arrange - create a Cart
    Cart cart = new Cart();

    // Arrange - create the controller
    CartController target = new CartController(null);

```

```

        // Act - call the Index action method
        CartIndexViewModel result
            = (CartIndexViewModel)target.Index(cart, "myUrl").ViewData.Model;

        // Assert
        Assert.AreSame(result.Cart, cart);
        Assert.AreEqual(result.ReturnUrl, "myUrl");
    }
}
}

```

---

## Completing the Cart

Now that I have introduced the custom model binder, it is time to complete the cart functionality by adding two new features. The first will allow the customer to remove an item from the cart. The second feature will display a summary of the cart at the top of the page.

### Removing Items from the Cart

I already defined and tested the `RemoveFromCart` action method in the controller, so letting the customer remove items is just a matter of exposing this method in a view, which I am going to do by adding a `Remove` button in each row of the cart summary. The changes to `Views/Cart/Index.cshtml` are shown in Listing 9-4.

**Listing 9-4.** Introducing a Remove Button to the `Index.cshtml` File

```

@model SportsStore.WebUI.Models.CartIndexViewModel

@{
    ViewBag.Title = "Sports Store: Your Cart";
}

<style>
    #cartTable td { vertical-align: middle; }
</style>

<h2>Your cart</h2>
<table id="cartTable" class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>

```

```

<tbody>
  @foreach (var line in Model.Cart.Lines) {
    <tr>
      <td class="text-center">@line.Quantity</td>
      <td class="text-left">@line.Product.Name</td>
      <td class="text-right">@line.Product.Price.ToString("c")</td>
      <td class="text-right">
        @((line.Quantity * line.Product.Price).ToString("c"))
      </td>
      <td>
        @using (Html.BeginForm("RemoveFromCart", "Cart")) {
          @Html.Hidden("ProductId", line.Product.ProductID)
          @Html.HiddenFor(x => x.ReturnUrl)
          <input class="btn btn-sm btn-warning"
            type="submit" value="Remove" />
        }
      </td>
    </tr>
  }
</tbody>
<tfoot>
  <tr>
    <td colspan="3" class="text-right">Total:</td>
    <td class="text-right">
      @Model.Cart.ComputeTotalValue().ToString("c")
    </td>
  </tr>
</tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>

```

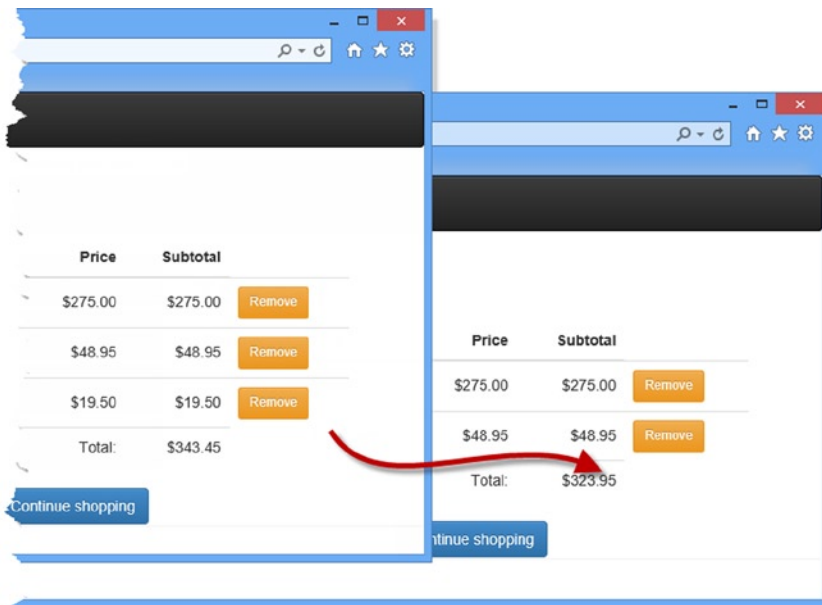
I added a new column to each row of the table that contains a form with an input element. I styled the input element as a button with Bootstrap and added a style element and an id to the table element to ensure that the button and the content of the other columns are properly aligned.

---

**Note** I used the strongly typed `Html.HiddenFor` helper method to create a hidden field for the `ReturnUrl` model property, but I had to use the string-based `Html.Hidden` helper to do the same for the `ProductId` field. If I had written `Html.HiddenFor(x => line.Product.ProductID)`, the helper would render a hidden field with the name `line.Product.ProductID`. The name of the field would not match the names of the parameters for the `CartController.RemoveFromCart` action method, which would prevent the default model binders from working, so the MVC Framework would not be able to call the method.

---

You can see the Remove buttons at work by running the application and adding items to the shopping cart. Remember that the cart already contains the functionality to remove it, which you can test by clicking one of the new buttons, as shown in Figure 9-1.



**Figure 9-1.** Removing an item from the shopping cart

## Adding the Cart Summary

I may have a functioning cart, but there is an issue with the way it is integrated into the interface. Customers can tell what is in their cart only by viewing the cart summary screen. And they can view the cart summary screen only by adding a new item to the cart.

To solve this problem, I am going to add a widget that summarizes the contents of the cart and that can be clicked to display the cart contents throughout the application. I will do this in much the same way that I added the navigation widget—as an action whose output I will inject into the Razor layout. To start, I need to add the simple method, shown in Listing 9-5, to the `CartController` class.

### **Listing 9-5.** Adding the Summary Method to the `CartController.cs` File

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

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

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

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



```

        public PartialViewResult Summary(Cart cart) {
            return PartialView(cart);
        }
    }
}

```

This simple method needs to render a view, supplying the current `Cart` (which will be obtained using the custom model binder) as view data. To create the view, right-click the `Summary` action method and select `Add View` from the pop-up menu. Set the name of the view to `Summary` and click the `OK` button to create the `Views/Cart/Summary.cshtml` file. Edit the view so that it matches Listing 9-6.

**Listing 9-6.** The Contents of the `Summary.cshtml` File

```

@model SportsStore.Domain.Entities.Cart

<div class="navbar-right">
    @Html.ActionLink("Checkout", "Index", "Cart",
        new { returnUrl = Request.Url.PathAndQuery },
        new { @class = "btn btn-default navbar-btn" })
</div>

<div class="navbar-text navbar-right">
    <b>Your cart:</b>
    @Model.Lines.Sum(x => x.Quantity) item(s),
    @Model.ComputeTotalValue().ToString("c")
</div>

```

The view displays the number of items in the cart, the total cost of those items, and a link that shows the details of the cart to the user (and, as you will have expected by now, I have assigned the elements that the view contains to classes defined by Bootstrap). Now that I have created the view that is returned by the `Summary` action method, I can call the `Summary` action method in the `_Layout.cshtml` file to display the cart summary, as shown in Listing 9-7.

**Listing 9-7.** Adding the `Summary` Partial View to the `_Layout.cshtml` File

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>

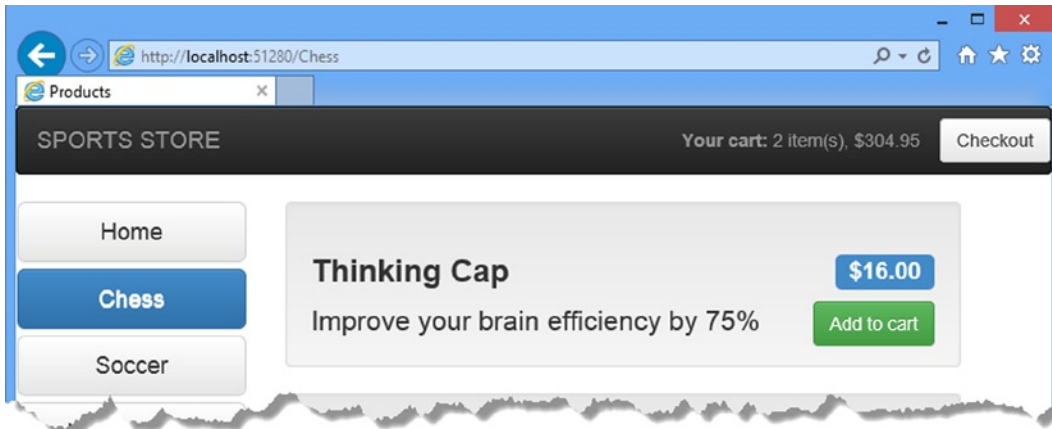
```

```

        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>

```

You can see the cart summary by running the application. The item count and total increase as you add items to the cart, as shown by Figure 9-2.



**Figure 9-2.** The cart summary widget

With this addition, customers know what is in their cart and have an obvious way to check out from the store. You can see, once again, how easy it is to use the `Html.Action` helper method to incorporate the output from an action method in another view. This is a nice technique for breaking down the functionality of an application into distinct, reusable blocks.

## Submitting Orders

I have now reached the final customer feature in SportsStore: the ability to check out and complete an order. In the following sections, I will extend the domain model to provide support for capturing the shipping details from a user and add the application support to process those details.

## Extending the Domain Model

Add a class file called `ShippingDetails.cs` to the `Entities` folder of the `SportsStore.Domain` project and edit it to match the contents shown in Listing 9-8. This is the class I will use to represent the shipping details for a customer.

**Listing 9-8.** The Contents of the `ShippingDetails.cs` File

```

using System.ComponentModel.DataAnnotations;

namespace SportsStore.Domain.Entities {

```

```

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

    [Required(ErrorMessage = "Please enter the first address line")]
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }

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

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

    public string Zip { get; set; }

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

    public bool GiftWrap { get; set; }
}

```

You can see that I am using the validation attributes from the `System.ComponentModel.DataAnnotations` namespace, just as I did in Chapter 2. I explore validation further in Chapter 25.

---

**Note** The `ShippingDetails` class does not have any functionality, so there is nothing that that can be sensibly unit tested.

---

## Adding the Checkout Process

The goal is to reach the point where users are able to enter their shipping details and submit their order. To start this off, I need to add a `Checkout now` button to the cart summary view. Listing 9-9 shows the change I applied to the `Views/Cart/Index.cshtml` file.

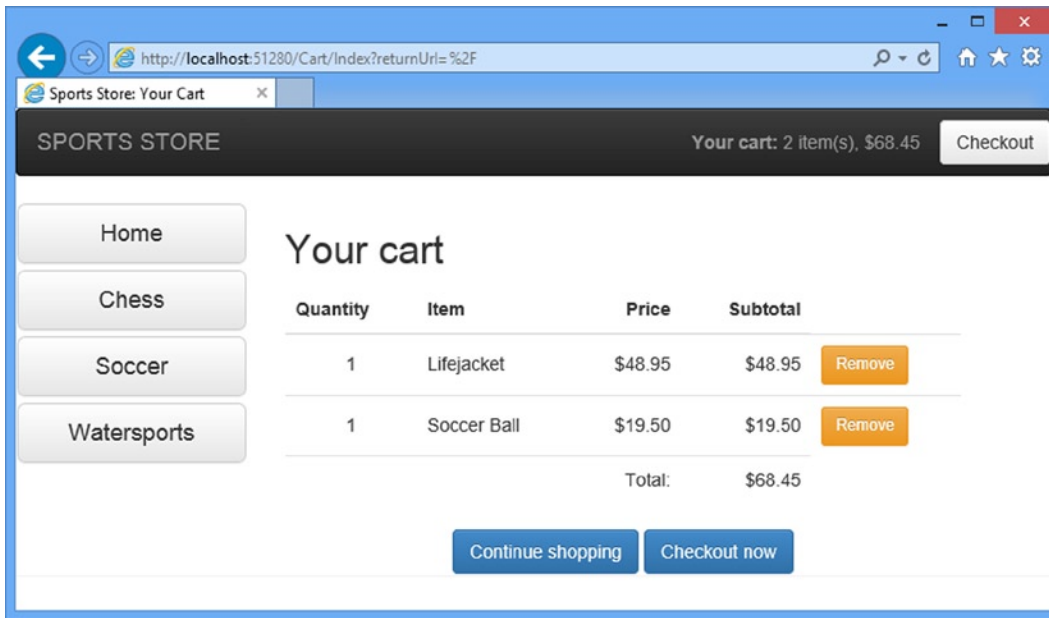
**Listing 9-9.** Adding the Checkout Now Button to the `Index.cshtml` File

```

...
<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
    @Html.ActionLink("Checkout now", "Checkout", null, new { @class = "btn btn-primary" })
</div>
...

```

This change generates a link that I have styled as a button and that, when clicked, calls the `Checkout` action method of the `Cart` controller. You can see how this button appears in Figure 9-3.



**Figure 9-3.** The Checkout now button

As you might expect, I now need to define the Checkout method in the CartController class, as shown in Listing 9-10.

**Listing 9-10.** The Checkout Action Method in the CartController.cs File

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

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

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

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

        public ActionResult Checkout() {
            return View(new ShippingDetails());
        }
    }
}
```

The Checkout method returns the default view and passes a new ShippingDetails object as the view model. To create the view for the action method, right-click on the Checkout action method and select Add View from the pop-up menu. Set the name to Checkout and click the OK button. Visual Studio will create the Views/Cart/Checkout.cshtml file, which you should edit to match Listing 9-11.

**Listing 9-11.** The Contents of the Checkout.cshtml File

```
@model SportsStore.Domain.Entities.ShippingDetails

@{
    ViewBag.Title = "SportStore: Checkout";
}

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

@using (Html.BeginForm()) {

    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name:</label>
        @Html.TextBoxFor(x => x.Name, new {@class = "form-control"})
    </div>

    <h3>Address</h3>

    <div class="form-group">
        <label>Line 1:</label>
        @Html.TextBoxFor(x => x.Line1, new {@class = "form-control"})
    </div>
    <div class="form-group">
        <label>Line 2:</label>
        @Html.TextBoxFor(x => x.Line2, new {@class = "form-control"})
    </div>
    <div class="form-group">
        <label>Line 3:</label>
        @Html.TextBoxFor(x => x.Line3, new {@class = "form-control"})
    </div>
    <div class="form-group">
        <label>City:</label>
        @Html.TextBoxFor(x => x.City, new {@class = "form-control"})
    </div>
    <div class="form-group">
        <label>State:</label>
        @Html.TextBoxFor(x => x.State, new {@class = "form-control"})
    </div>
    <div class="form-group">
        <label>Zip:</label>
        @Html.TextBoxFor(x => x.Zip, new {@class = "form-control"})
    </div>
}
```

```

<div class="form-group">
  <label>Country:</label>
  @Html.TextBoxFor(x => x.Country, new {@class = "form-control"})
</div>

<h3>Options</h3>
<div class="checkbox">
  <label>
    @Html.EditorFor(x => x.GiftWrap)
    Gift wrap these items
  </label>
</div>

<div class="text-center">
  <input class="btn btn-primary" type="submit" value="Complete order" />
</div>
}

```

For each of the properties in the model, I have created a label and input element formatted with Bootstrap to capture the user input. You can see the effect I have created by starting the application and clicking the Checkout button at the top of the page and then clicking Checkout now, as shown by Figure 9-4. (You can also reach this view by navigating to the /Cart/Checkout URL).

**Figure 9-4.** The shipping details form

The problem with this view is that it contains a lot of repeated markup. There are MVC Framework HTML helpers that could reduce the duplication, but they make it hard to structure and style the content in the way that I want. Instead, I am going to use a handy feature to get metadata about the view model object and combine it with a mix of C# and Razor expressions. You can see what I have done in Listing 9-12.

**Listing 9-12.** Reducing Duplication in the Checkout.cshtml File

```
@model SportsStore.Domain.Entities.ShippingDetails

@{
    ViewBag.Title = "SportStore: Checkout";
}
```

```

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

@using (Html.BeginForm()) {

    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.Name, new {@class = "form-control"})
    </div>

    <h3>Address</h3>

    foreach (var property in ViewData.ModelMetadata.Properties) {
        if (property.PropertyName != "Name" && property.PropertyName != "GiftWrap") {
            <div class="form-group">
                <label>@(property.DisplayName ?? property.PropertyName)</label>
                @Html.TextBox(property.PropertyName, null, new {@class = "form-control"})
            </div>
        }
    }

    <h3>Options</h3>
    <div class="checkbox">
        <label>
            @Html.EditorFor(x => x.GiftWrap)
            Gift wrap these items
        </label>
    </div>

    <div class="text-center">
        <input class="btn btn-primary" type="submit" value="Complete order" />
    </div>
}

```

The static `ViewData.ModelMetadata` property returns a `System.Web.Mvc.ModelMetadata` object that provides information about the model type for the view. The `Properties` property I use in the `foreach` loop returns a collection of `ModelMetadata` objects, each of which represents a property defined by the model type. I use the `PropertyName` property to ensure that I don't generate content for the `Name` or `GiftWrap` properties (which I deal with elsewhere in the view) and generate a set of elements, complete with Bootstrap classes, for all of the other properties.

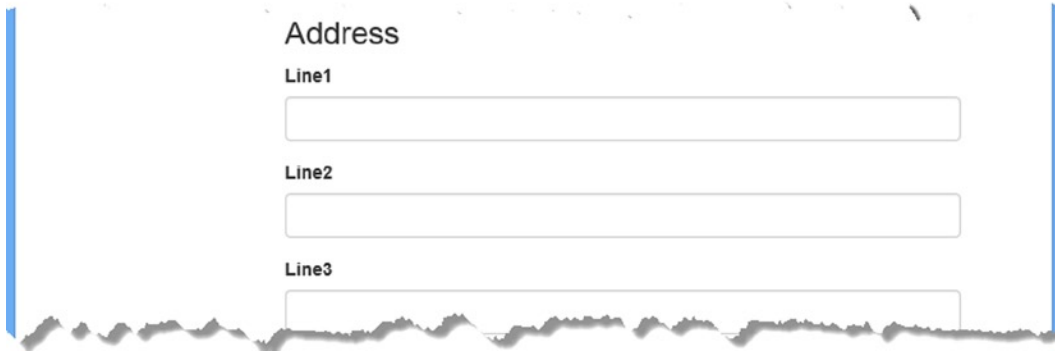
---

■ **Tip** The `for` and `if` keywords I have used are within the scope of a Razor expression (the `@using` expression that creates the form) and so I don't need to prefix them with the `@` character. In fact, were I to do so, Razor would report an error. It can take a little while to get used to when the `@` character is required with Razor, but it becomes second nature for most programmers. For those that can't quite get it right first time (which includes me), the Razor error message displayed in the browser provides specific instructions to correct any mistakes.

---



I am not quite done, however. If you run the example and look at the output generated by the view, you will see that some of the labels are not quite correct, as Figure 9-5 illustrates.



**Figure 9-5.** The problem with generating labels from property names

The issue is that the property names don't always make for good labels. This is why I check to see if there is a `DisplayName` value available when I generate the form elements, like this:

```
...
<label>@(property.DisplayName ?? property.PropertyName)</label>
...
```

To take advantage of the `DisplayName` property, I need to apply the `Display` attribute to the model class, as shown in Listing 9-13.

**Listing 9-13.** Applying the `Display` attribute to the `ShippingDetails.cs` File

```
using System.ComponentModel.DataAnnotations;
```

```
namespace SportsStore.Domain.Entities {

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

        [Required(ErrorMessage = "Please enter the first address line")]
        [Display(Name="Line 1")]
        public string Line1 { get; set; }
        [Display(Name = "Line 2")]
        public string Line2 { get; set; }
        [Display(Name = "Line 3")]
        public string Line3 { get; set; }

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

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

```

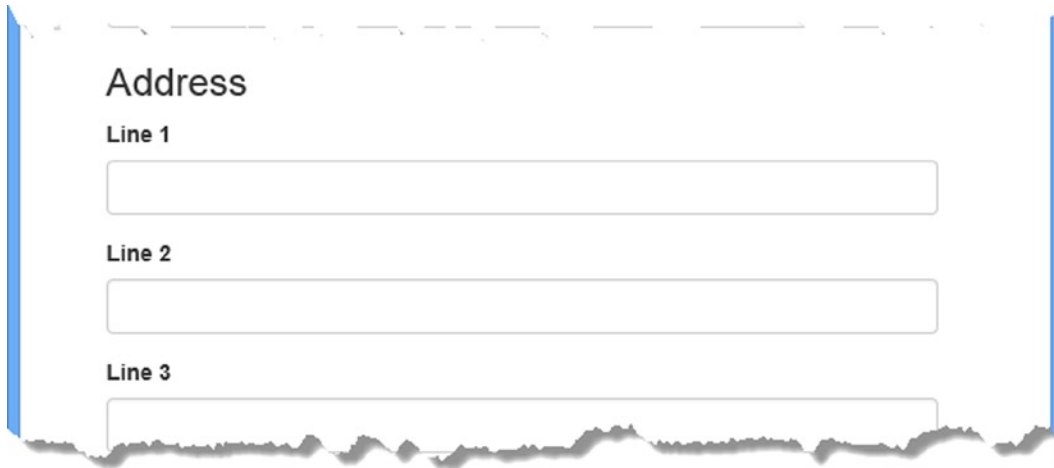
    public string Zip { get; set; }

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

    public bool GiftWrap { get; set; }
}

```

Setting the Name value for the Display attribute allows me to set up a value that will be read by the DisplayName property in the view. You can see the effect by starting the application and viewing the checkout page, as shown in Figure 9-6.



The screenshot shows a web form titled "Address" with three input fields. The first field is labeled "Line 1", the second "Line 2", and the third "Line 3". Each field is a simple text box with a light gray border. The form is set against a light gray background with a subtle grid pattern. The entire form is framed by a blue border on the left and right sides.

**Figure 9-6.** The effect of the Display attribute on the model type

This example shows two different aspects of working with the MVC Framework. The first is that you can work around any situation to simplify your markup or code. The second is that even though the role that views play in the MVC pattern is restricted to displaying data and markup, the tools that Razor and C# provide for this task are rich and flexible, even to the extent of working with type metadata.

## Implementing the Order Processor

I need a component in the application to which I can hand details of an order for processing. In keeping with the principles of the MVC model, I am going to define an interface for this functionality, write an implementation of the interface, and then associate the two using the DI container, Ninject.

### Defining the Interface

Add a new interface called `IOrderProcessor` to the `Abstract` folder of the `SportsStore.Domain` project and edit the contents so that they match Listing 9-14.

**Listing 9-14.** The Contents of the IOrderProcessor.cs File

```
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {

    public interface IOrderProcessor {

        void ProcessOrder(Cart cart, ShippingDetails shippingDetails);

    }

}
```

## Implementing the Interface

The implementation of `IOrderProcessor` is going to deal with orders by e-mailing them to the site administrator. I am simplifying the sales process, of course. Most e-commerce sites would not simply e-mail an order, and I have not provided support for processing credit cards or other forms of payment. But I want to keep things focused on MVC, and so e-mail it is.

Create a new class file called `EmailOrderProcessor.cs` in the `Concrete` folder of the `SportsStore.Domain` project and edit the contents so that they match Listing 9-15. This class uses the built-in SMTP support included in the .NET Framework library to send an e-mail.

**Listing 9-15.** The Contents of the EmailOrderProcessor.cs File

```
using System.Net;
using System.Net.Mail;
using System.Text;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Concrete {

    public class EmailSettings {
        public string MailToAddress = "orders@example.com";
        public string MailFromAddress = "sportsstore@example.com";
        public bool UseSsl = true;
        public string Username = "MySmtpUsername";
        public string Password = "MySmtpPassword";
        public string ServerName = "smtp.example.com";
        public int ServerPort = 587;
        public bool WriteAsFile = false;
        public string FileLocation = @"c:\sports_store_emails";
    }

    public class EmailOrderProcessor : IOrderProcessor {
        private EmailSettings emailSettings;

        public EmailOrderProcessor(EmailSettings settings) {
            emailSettings = settings;
        }
    }
}
```

```

public void ProcessOrder(Cart cart, ShippingDetails shippingInfo) {

    using (var smtpClient = new SmtpClient()) {

        smtpClient.EnableSsl = emailSettings.UseSsl;
        smtpClient.Host = emailSettings.ServerName;
        smtpClient.Port = emailSettings.ServerPort;
        smtpClient.UseDefaultCredentials = false;
        smtpClient.Credentials
            = new NetworkCredential(emailSettings.Username,
                                    emailSettings.Password);

        if (emailSettings.WriteAsFile) {
            smtpClient.DeliveryMethod
                = SmtpDeliveryMethod.SpecifiedPickupDirectory;
            smtpClient.PickupDirectoryLocation = emailSettings.FileLocation;
            smtpClient.EnableSsl = false;
        }

        StringBuilder body = new StringBuilder()
            .AppendLine("A new order has been submitted")
            .AppendLine("---")
            .AppendLine("Items:");

        foreach (var line in cart.Lines) {
            var subtotal = line.Product.Price * line.Quantity;
            body.AppendFormat("{0} x {1} (subtotal: {2:c})", line.Quantity,
                           line.Product.Name,
                           subtotal);
        }

        body.AppendFormat("Total order value: {0:c}", cart.ComputeTotalValue())
            .AppendLine("---")
            .AppendLine("Ship to:")
            .AppendLine(shippingInfo.Name)
            .AppendLine(shippingInfo.Line1)
            .AppendLine(shippingInfo.Line2 ?? "")
            .AppendLine(shippingInfo.Line3 ?? "")
            .AppendLine(shippingInfo.City)
            .AppendLine(shippingInfo.State ?? "")
            .AppendLine(shippingInfo.Country)
            .AppendLine(shippingInfo.Zip)
            .AppendLine("---")
            .AppendFormat("Gift wrap: {0}",
                          shippingInfo.GiftWrap ? "Yes" : "No");

        MailMessage mailMessage = new MailMessage(
            emailSettings.MailFromAddress, // From
            emailSettings.MailToAddress,   // To
            "New order submitted!",         // Subject
            body.ToString());              // Body
    }
}

```

```

        if (emailSettings.WriteAsFile) {
            mailMessage.BodyEncoding = Encoding.ASCII;
        }

        smtpClient.Send(mailMessage);
    }
}
}
}

```

To make things simpler, I have defined the `EmailSettings` class in Listing 9-15 as well. An instance of this class is demanded by the `EmailOrderProcessor` constructor and contains all the settings that are required to configure the .NET e-mail classes.

---

■ **Tip** Do not worry if you do not have an SMTP server available. If you set the `EmailSettings.WriteAsFile` property to `true`, the e-mail messages will be written as files to the directory specified by the `FileLocation` property. This directory must exist and be writable. The files will be written with the `.eml` extension, but they can be read with any text editor. The location I have set in the listing is `c:\sports_store_emails`.

---

## Registering the Implementation

Now that I have an implementation of the `IOrderProcessor` interface and the means to configure it, I can use Ninject to create instances of it. Edit the `NinjectDependencyResolver.cs` file in the `Infrastructure` folder of the `SportsStore.WebUI` project and make the changes shown in Listing 9-16 to the `AddBindings` method.

**Listing 9-16.** Adding Ninject Bindings for `IOrderProcessor` to 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;

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);
}
}
}

```

I created an `EmailSettings` object, which I use with the `Ninject WithConstructorArgument` method so that I can inject it into the `EmailOrderProcessor` constructor when new instances are created to service requests for the `IOrderProcessor` interface. In Listing 9-16, I specified a value for only one of the `EmailSettings` properties: `WriteAsFile`. I read the value of this property using the `ConfigurationManager.AppSettings` property, which provides access to application settings defined in the `Web.config` file (the one in the root project folder), which are shown in Listing 9-17.

**Listing 9-17.** Application Settings in the `Web.config` File

```

...
<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>
...

```

## Completing the Cart Controller

To complete the `CartController` class, I need to modify the constructor so that it demands an implementation of the `IOrderProcessor` interface and add a new action method that will handle the HTTP form POST request when the user clicks the Complete order button. Listing 9-18 shows both changes.

**Listing 9-18.** Completing the Controller in the `CartController.cs` File

```

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

```

```

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private IOrderProcessor orderProcessor;

        public CartController(IProductRepository repo, IOrderProcessor proc) {
            repository = repo;
            orderProcessor = proc;
        }

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

        public ActionResult Checkout() {
            return View(new ShippingDetails());
        }

        [HttpPost]
        public ActionResult Checkout(Cart cart, ShippingDetails shippingDetails) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }

            if (ModelState.IsValid) {
                orderProcessor.ProcessOrder(cart, shippingDetails);
                cart.Clear();
                return View("Completed");
            } else {
                return View(shippingDetails);
            }
        }
    }
}

```

You can see that the Checkout action method I added is decorated with the HttpPost attribute, which means that it will be invoked for a POST request—in this case, when the user submits the form. Once again, I am relying on the model binder system, both for the ShippingDetails parameter (which is created automatically using the HTTP form data) and the Cart parameter (which is created using the custom binder).

---

■ **Note** The change in constructor forces me to update the unit tests I created for the CartController class. Passing null for the new constructor parameter will let the unit tests compile.

---

The MVC Framework checks the validation constraints that I applied to ShippingDetails using the data annotation attributes, and any validation problems violations are passed to the action method through the ModelState property. I can see if there are any problems by checking the ModelState.IsValid property. Notice that I call the ModelState.AddModelError method to register an error message if there are no items in the cart. I will explain how to display such errors shortly, and I have much more to say about model binding and validation in Chapters 24 and 25.

## UNIT TEST: ORDER PROCESSING

To complete the unit testing for the `CartController` class, I need to test the behavior of the new overloaded version of the `Checkout` method. Although the method looks short and simple, the use of MVC Framework model binding means that there is a lot going on behind the scenes that needs to be tested.

I want to process an order only if there are items in the cart *and* the customer has provided valid shipping details. Under all other circumstances, the customer should be shown an error. Here is the first test method:

```
...
[TestMethod]
public void Cannot_Checkout_Empty_Cart() {

    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();
    // Arrange - create an empty cart
    Cart cart = new Cart();
    // Arrange - create shipping details
    ShippingDetails shippingDetails = new ShippingDetails();
    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Act
    ViewResult result = target.Checkout(cart, shippingDetails);

    // Assert - check that the order hasn't been passed on to the processor
    mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()),
        Times.Never());
    // Assert - check that the method is returning the default view
    Assert.AreEqual("", result.ViewName);
    // Assert - check that I am passing an invalid model to the view
    Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
...
```

This test ensures that I cannot check out with an empty cart. I check this by ensuring that the `ProcessOrder` of the mock `IOrderProcessor` implementation is never called, that the view that the method returns is the default view (which will redisplay the data entered by customers and give them a chance to correct it), and that the model state being passed to the view has been marked as invalid. This may seem like a belt-and-braces set of assertions, but I need all three to be sure that I have the right behavior. The next test method works in much the same way, but injects an error into the view model to simulate a problem reported by the model binder (which would happen in production when the customer enters invalid shipping data):

```
...
[TestMethod]
public void Cannot_Checkout_Invalid_ShippingDetails() {

    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();
```



```

// Arrange - create a cart with an item
Cart cart = new Cart();
cart.AddItem(new Product(), 1);

// Arrange - create an instance of the controller
CartController target = new CartController(null, mock.Object);
// Arrange - add an error to the model
target.ModelState.AddModelError("error", "error");

// Act - try to checkout
ActionResult result = target.Checkout(cart, new ShippingDetails());

// Assert - check that the order hasn't been passed on to the processor
mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()),
    Times.Never());
// Assert - check that the method is returning the default view
Assert.AreEqual("", result.ViewName);
// Assert - check that I am passing an invalid model to the view
Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
...

```

Having established that an empty cart or invalid details will prevent an order from being processed, I need to ensure that I process orders when appropriate. Here is the test:

```

...
[TestMethod]
public void Can_Checkout_And_Submit_Order() {
    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();
    // Arrange - create a cart with an item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Act - try to checkout
    ActionResult result = target.Checkout(cart, new ShippingDetails());

    // Assert - check that the order has been passed on to the processor
    mock.Verify(m => m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()),
        Times.Once());
    // Assert - check that the method is returning the Completed view
    Assert.AreEqual("Completed", result.ViewName);
    // Assert - check that I am passing a valid model to the view
    Assert.AreEqual(true, result.ViewData.ModelState.IsValid);
}
...

```

Notice that I did not need to test that I can identify valid shipping details. This is handled for me automatically by the model binder using the attributes applied to the properties of the `ShippingDetails` class.

## Displaying Validation Errors

The MVC Framework will use the validation attributes applied to the `ShippingDetails` class to validate user data. However, I need to make a couple of changes to display any problems to the user. The first issue is that I need to provide a summary of any problems to the user. This is especially important when dealing with problems that are not related to specific fields, such as when the user tries to check out with no items in the cart.

To display a useful summary of the validation errors, I can use the `Html.ValidationSummary` helper method, just as I did in Chapter 2. Listing 9-19 shows the addition to `Checkout.cshtml` view.

**Listing 9-19.** Adding a Validation Summary to the `Checkout.cshtml` File

```
...
@using (Html.BeginForm()) {

    @Html.ValidationSummary()
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name</label>
        @Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
    </div>

    <h3>Address</h3>
    ...
}
```

The next step is to create some CSS styles that target the classes used by the validation summary and those to which the MVC Framework adds invalid elements. I created a new Style Sheet called `ErrorStyles.css` in the `Content` folder of the `SportsStore.WebUI` project and defined the styles shown in Listing 9-20. This is the same set of styles that I used in Chapter 2.

**Listing 9-20.** The Contents of the `ErrorStyles.css` File

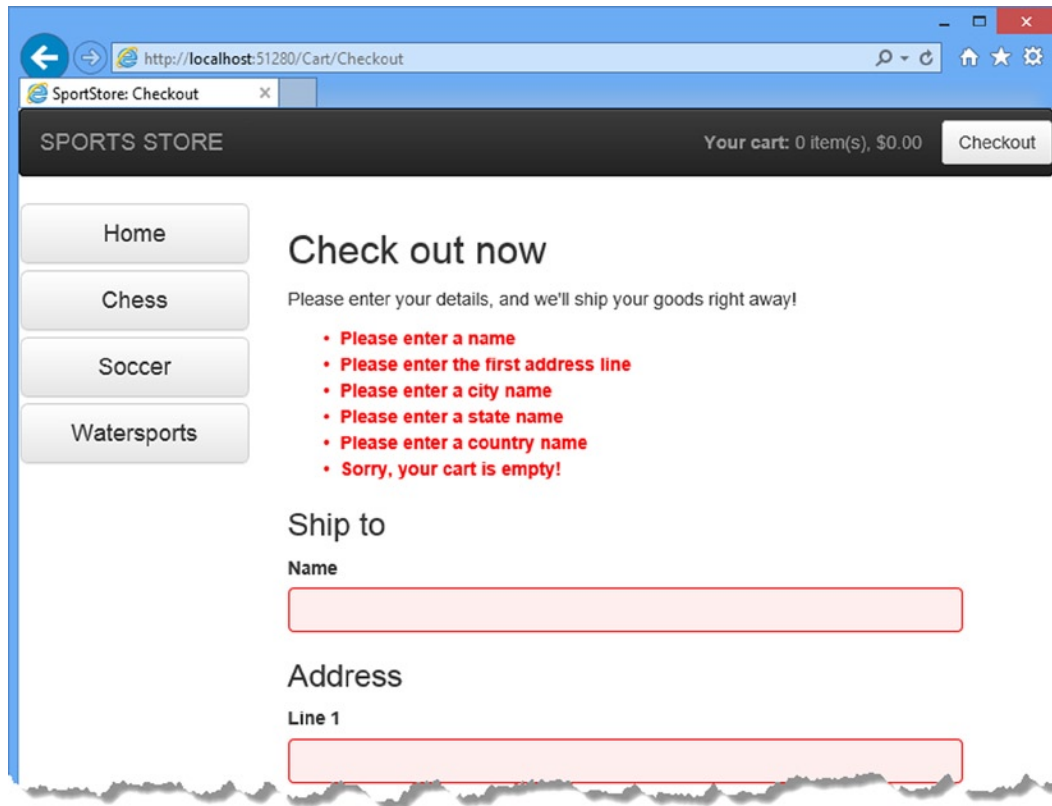
```
.field-validation-error    { color: #f00; }
.field-validation-valid    { display: none; }
.input-validation-error    { border: 1px solid #f00; background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00; }
.validation-summary-valid  { display: none; }
```

In order to use apply the styles, I updated the `_Layout.cshtml` file to add a link element for the `ErrorStyles.css` file, as shown in Listing 9-21.

**Listing 9-21.** Adding a Link Element in the `_Layout.cshtml` File

```
...
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
...
```

With these changes, validation errors are reported through highlighting problematic fields and by showing a summary of problems, as Figure 9-7 illustrates.



**Figure 9-7.** *Displaying validation messages*

---

■ **Tip** The data submitted by the user is sent to the server before it is validated, which is known as server-side validation and for which the MVC Framework has excellent support. The problem with server-side validation is that the user isn't told about errors until after the data has been sent to the server, processed and the result page generated—something that can take a few seconds on a busy server. For this reason, server-side validation is usually complemented by client-side validation, where JavaScript is used to check the values that the user has entered before the form data is sent to the server. I describe client-side validation in Chapter 25.

---

## Displaying a Summary Page

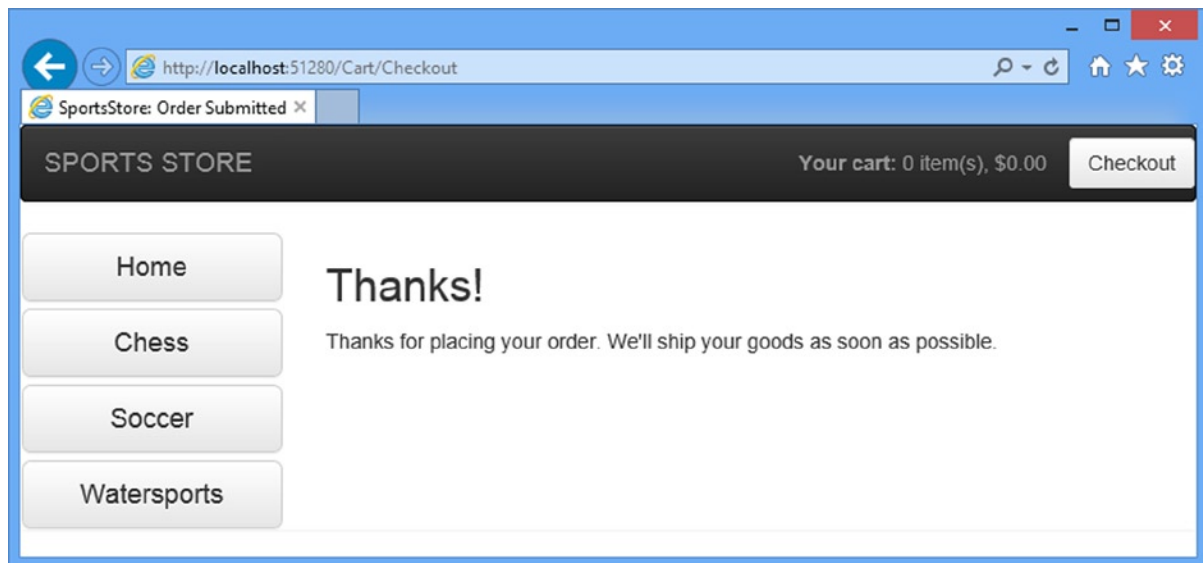
To complete the checkout process, I will show customers a page that confirms the order has been processed and thanks them for their business. Create a new view called `Completed.cshtml` in the `Views/Cart` folder and edit the content to match Listing 9-22.

**Listing 9-22.** The Contents of the `Completed.cshtml` File

```
@{
    ViewBag.Title = "SportsStore: Order Submitted";
}

<h2>Thanks!</h2>
Thanks for placing your order. We'll ship your goods as soon as possible.
```

I don't need to make any code changes to integrate this view into the application because I already added the required statements when I defined the Checkout action method back in Listing 9-18. Now customers can go through the entire process, from selecting products to checking out. If they provide valid shipping details (and have items in their cart), they will see the summary page when they click the Complete order button, as shown in Figure 9-8.



**Figure 9-8.** The thank-you page

## Summary

I have completed all the major parts of the customer-facing portion of SportsStore. It might not be enough to worry Amazon, but I have a product catalog that can be browsed by category and page, a neat shopping cart, and a simple checkout process.

The well-separated architecture means I can easily change the behavior of any piece of the application without worrying about causing problems or inconsistencies elsewhere. For example, I could process orders by storing them in a database, and it would not have any impact on the shopping cart, the product catalog, or any other area of the application. In the next chapter, I use two different techniques to create a mobile version of the SportsStore application.