

CHAPTER 10



SportsStore: Completing the Cart

In this chapter, I continue to build 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.

Refining the Cart Model with a Service

I defined a `Cart` model class in the previous chapter and demonstrated how it can be stored using the session feature, allowing the user to build up a set of products for purchase. The responsibility for managing the persistence of the `Cart` class fell to the `Cart` controller, which explicitly defines methods for getting and storing `Cart` objects.

The problem with this approach is that I will have to duplicate the code that obtains and stores `Cart` objects in any component that uses them. In this section, I am going to use the services feature that sits at the heart of ASP.NET Core to simplify the way that `Cart` objects are managed, freeing individual components such as the `Cart` controller from needing to deal with the details directly.

Services are most commonly used to hide details of how interfaces are implemented from the components that depend on them. You saw an example of this when I created a service for the `IProductRepository` interface, which allowed me to seamlessly replace the fake repository class with the Entity Framework Core repository. But services can be used to solve lots of other problems as well and can be used to shape and reshape an application, even when you are working with concrete classes such as `Cart`.

Creating a Storage-Aware Cart Class

The first step in tidying up the way that the `Cart` class is used will be to create a subclass that is aware of how to store itself using session state. I added a class file called `SessionCart.cs` to the `Models` folder and used it to define the class shown in Listing 10-1.

Listing 10-1. The Contents of the `SessionCart.cs` File in the `Models` Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using SportsStore.Infrastructure;

namespace SportsStore.Models {

    public class SessionCart: Cart {
```

```

public static Cart GetCart(IServiceProvider services) {
    ISession session = services.GetRequiredService<IHttpContextAccessor>()?
        .HttpContext.Session;
    SessionCart cart = session?.GetJson<SessionCart>("Cart")
        ?? new SessionCart();
    cart.Session = session;
    return cart;
}

[JsonIgnore]
public ISession Session { get; set; }

public override void AddItem(Product product, int quantity) {
    base.AddItem(product, quantity);
    Session.SetJson("Cart", this);
}

public override void RemoveLine(Product product) {
    base.RemoveLine(product);
    Session.SetJson("Cart", this);
}

public override void Clear() {
    base.Clear();
    Session.Remove("Cart");
}
}
}

```

The `SessionCart` class subclasses the `Cart` class and overrides the `AddItem`, `RemoveLine`, and `Clear` methods so they call the base implementations and then store the updated state in the session using the extension methods on the `ISession` interface I defined in Chapter 9. The static `GetCart` method is a factory for creating `SessionCart` objects and providing them with an `ISession` object so they can store themselves.

Getting hold of the `ISession` object is a little complicated. I have to obtain an instance of the `IHttpContextAccessor` service, which provides me with access to an `HttpContext` object that, in turn, provides me with the `ISession`. This indirect approach is required because the session isn't provided as a regular service.

Registering the Service

The next step is to create a service for the `Cart` class. My goal is to satisfy requests for `Cart` objects with `SessionCart` objects that will seamlessly store themselves. You can see how I created the service in Listing 10-2.

Listing 10-2. Creating the `Cart` Service in the `Startup.cs` File in the `SportsStore` Folder

```

...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration["Data:SportStoreProducts:ConnectionString"]));
    services.AddTransient<IProductRepository, EFProductRepository>();
}

```

```

services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
services.AddMvc();
services.AddMemoryCache();
services.AddSession();
}
...

```

The `AddScoped` method specifies that the same object should be used to satisfy related requests for `Cart` instances. How requests are related can be configured, but by default, it means that any `Cart` required by components handling the same HTTP request will receive the same object.

Rather than provide the `AddScoped` method with a type mapping, as I did for the repository, I have specified a lambda expression that will be invoked to satisfy `Cart` requests. The expression receives the collection of services that have been registered and passes the collection to the `GetCart` method of the `SessionCart` class. The result is that requests for the `Cart` service will be handled by creating `SessionCart` objects, which will serialize themselves as session data when they are modified.

I also added a service using the `AddSingleton` method, which specifies that the same object should always be used. The service I created tells MVC to use the `HttpContextAccessor` class when implementations of the `IHttpContextAccessor` interface are required. This service is required so I can access the current session in the `SessionCart` class in Listing 10-1.

Simplifying the Cart Controller

The benefit of creating this kind of service is that it allows me to simplify the controllers where `Cart` objects are used. In Listing 10-3, I have reworked the `CartController` class to take advantage of the new service.

Listing 10-3. Using the `Cart` Service in the `CartController.cs` File in the `Controllers` Folder

```

using System.Linq;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private Cart cart;

        public CartController(IProductRepository repo, Cart cartService) {
            repository = repo;
            cart = cartService;
        }

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

```

```

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

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

```

The `CartController` class indicates that it needs a `Cart` object by declaring a constructor argument, which has allowed me to remove the methods that read and write data from the session and the steps required to write updates. The result is a controller that is simpler and remains focused on its role in the application without having to worry about how `Cart` objects are created or persisted. And, since services are available throughout the application, any component can get hold of the user's cart using the same technique.

Completing the Cart Functionality

Now that I have introduced the `Cart` service, 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. Listing 10-4 shows the changes to the `Views/Cart/Index.cshtml` file.

Listing 10-4. Introducing a `Remove` Button to the `Index.cshtml` File in the `Views/Cart` Folder

```

@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
  <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>
                <form asp-action="RemoveFromCart" method="post">
                    <input type="hidden" name="ProductID"
                        value="@line.Product.ProductID" />
                    <input type="hidden" name="returnUrl"
                        value="@Model.ReturnUrl" />
                    <button type="submit" class="btn btn-sm btn-danger">
                        Remove
                    </button>
                </form>
            </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 hidden input elements that specify the product to be removed and the return URL, along with a button that submits the form.

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 10-1.

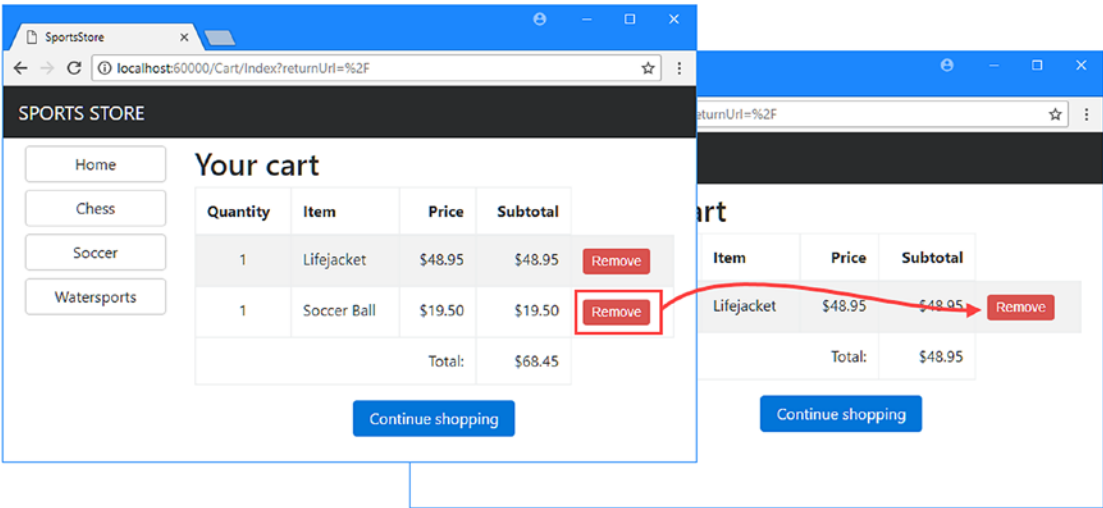


Figure 10-1. Removing an item from the shopping cart

Adding the Cart Summary Widget

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 a view component whose output I can include in the Razor shared layout.

Adding the Font Awesome Package

As part of the cart summary, I am going to display a button that allows the user to check out. Rather than display the word *checkout* in the button, I want to use a cart symbol. Since I have no artistic skills, I am going to use the Font Awesome package, which is an excellent set of open source icons that are integrated into applications as fonts, where each character in the font is a different image. You can learn more about Font Awesome, including inspecting the icons it contains, at <http://fontawesome.github.io/Font-Awesome>.

I selected the SportsStore project and clicked the Show All Items button at the top of the Solution Explorer to reveal the `bower.json` file. I then added the Font Awesome package to the dependencies section, as shown in Listing 10-5.

Listing 10-5. Adding the Font Awesome Package in the `bower.json` File in the SportsStore Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0"
  }
}
```

When the `bower.json` file is saved, Visual Studio uses Bower to download and install the Font Awesome package in the `www/lib/fontawesome` folder.

Creating the View Component Class and View

I added a class file called `CartSummaryViewComponent.cs` in the `Components` folder and used it to define the view component shown in Listing 10-6.

Listing 10-6. The Contents of the `CartSummaryViewComponent.cs` File in the `Components` Folder

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

namespace SportsStore.Components {

    public class CartSummaryViewComponent : ViewComponent {
        private Cart cart;

        public CartSummaryViewComponent(Cart cartService) {
            cart = cartService;
        }

        public IViewComponentResult Invoke() {
            return View(cart);
        }
    }
}
```

This view component is able to take advantage of the service that I created earlier in the chapter in order to receive a `Cart` object as a constructor argument. The result is a simple view component class that passes on the `Cart` object to the `View` method in order to generate the fragment of HTML that will be included in the layout. To create the layout, I created the `Views/Shared/Components/CartSummary` folder, added to it a Razor view file called `Default.cshtml`, and added the markup shown in Listing 10-7.

Listing 10-7. The `Default.cshtml` File in the `Views/Shared/Components/CartSummary` Folder

```
@model Cart

<div class="">
    @if (Model.Lines.Count() > 0) {
        <small class="navbar-text">
            <b>Your cart:</b>
            @Model.Lines.Sum(x => x.Quantity) item(s)
            @Model.ComputeTotalValue().ToString("c")
        </small>
    }
    <a class="btn btn-sm btn-secondary navbar-btn"
        asp-controller="Cart" asp-action="Index"
        asp-route-returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">
        <i class="fa fa-shopping-cart"></i>
    </a>
</div>
```

The view displays a button with the Font Awesome cart icon and, if there are items in the cart, provides a snapshot that details the number of items and their total value. Now that I have a view component and a view, I can modify the shared layout so that the cart summary is included in the responses generated by the application's controllers, as shown in Listing 10-8.

Listing 10-8. Adding the Cart Summary in the `_Layout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet"
    asp-href-include="/lib/bootstrap/dist/**/*.min.css"
    asp-href-exclude="**/*-reboot*,**/*-grid*" />
  <link rel="stylesheet" asp-href-include="/lib/fontawesome/css/*.css" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <div class="row">
      <a class="col navbar-brand" href="#">SPORTS STORE</a>
      <div class="col-4 text-right">
        @await Component.InvokeAsync("CartSummary")
      </div>
    </div>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      @await Component.InvokeAsync("NavigationMenu")
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

You can see the cart summary by starting the application. When the cart is empty, only the checkout button is shown. If you add items to the cart, then the number of items and their combined cost are shown, as illustrated in Figure 10-2. With this addition, customers know what is in their cart and have an obvious way to check out from the store.

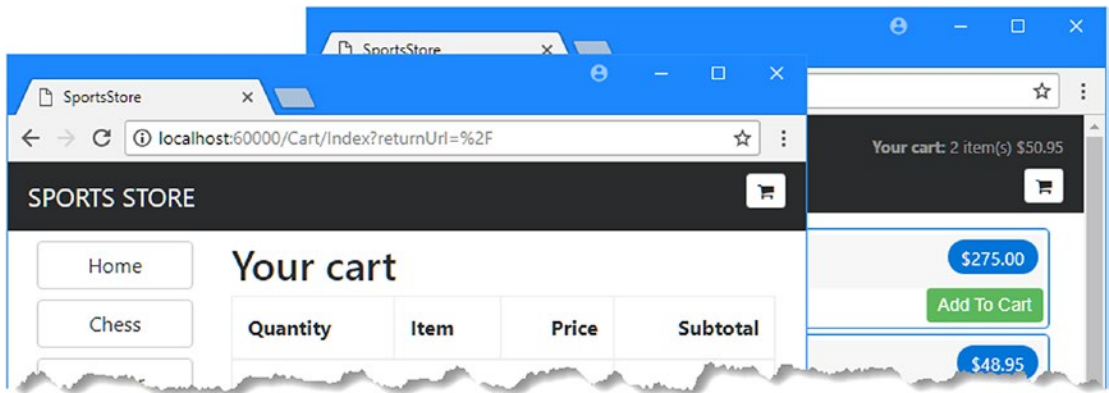


Figure 10-2. *Displaying a summary of the cart*

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.

Creating the Model Class

I added a class file called `Order.cs` to the `Models` folder and edited it to match the contents shown in Listing 10-9. This is the class I will use to represent the shipping details for a customer.

Listing 10-9. The Contents of the `Order.cs` File in the `Models` Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Order {

        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [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; }
}
}

```

I am using the validation attributes from the `System.ComponentModel.DataAnnotations` namespace, just as I did in Chapter 2. I describe validation further in Chapter 27.

I also use the `BindNever` attribute, which prevents the user from supplying values for these properties in an HTTP request. This is a feature of the model binding system, which I describe in Chapter 26; it stops MVC using values from the HTTP request to populate sensitive or important model properties.

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, I need to add a Checkout button to the cart summary view. Listing 10-10 shows the change I applied to the `Views/Cart/Index.cshtml` file.

Listing 10-10. Adding the Checkout Now Button to the `Index.cshtml` File in the `Views/Cart` Folder

```

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

```

This change generates a link that I have styled as a button and that, when clicked, calls the `Checkout` action method of the `Order` controller, which I create in the following section. You can see how this button appears in Figure 10-3.

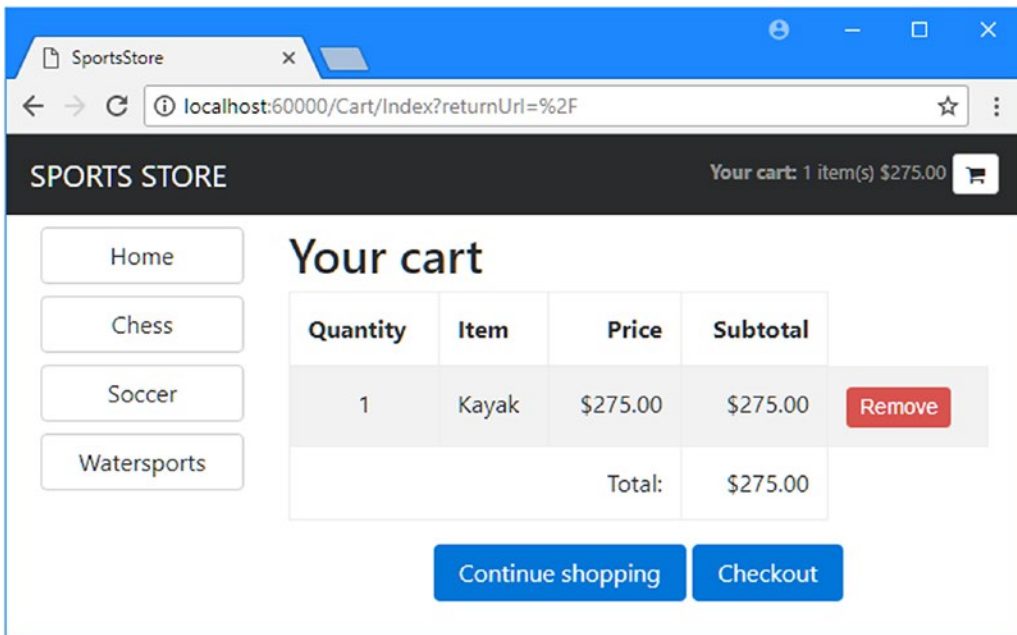


Figure 10-3. The Checkout button

I now need to define the Order controller. I added a class file called `OrderController.cs` to the `Controllers` folder and used it to define the class shown in Listing 10-11.

Listing 10-11. The Contents of the `OrderController.cs` File in the `Controllers` Folder

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

namespace SportsStore.Controllers {

    public class OrderController : Controller {

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

    }

}
```

The `Checkout` method returns the default view and passes a new `ShippingDetails` object as the view model. To create the view, I created the `Views/Order` folder and added a Razor view file called `Checkout.cshtml` with the markup shown in Listing 10-12.

Listing 10-12. The Contents of the Checkout.cshtml File in the Views/Order Folder

```

@model Order

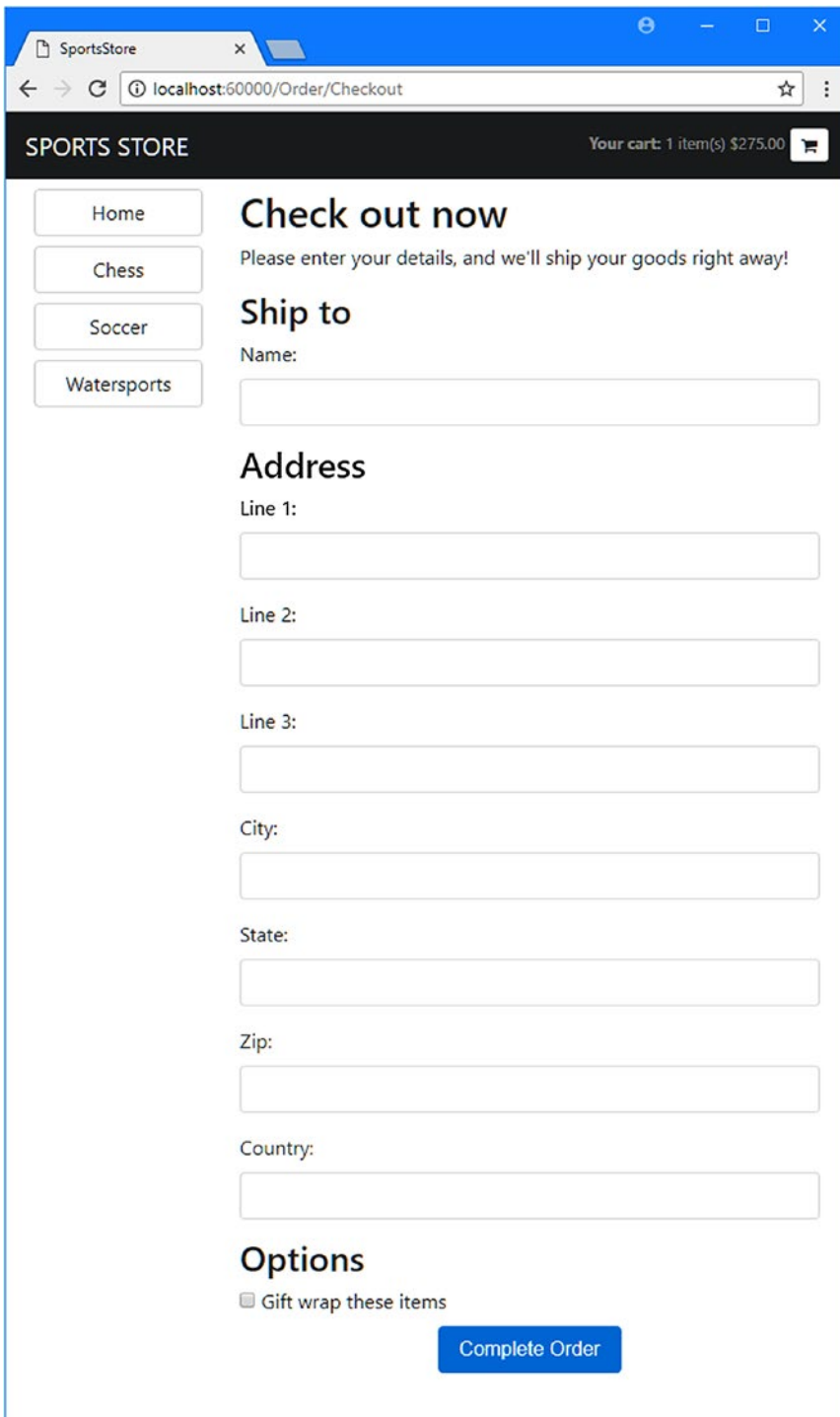
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form asp-action="Checkout" method="post">
  <h3>Ship to</h3>
  <div class="form-group">
    <label>Name:</label><input asp-for="Name" class="form-control" />
  </div>
  <h3>Address</h3>
  <div class="form-group">
    <label>Line 1:</label><input asp-for="Line1" class="form-control" />
  </div>
  <div class="form-group">
    <label>Line 2:</label><input asp-for="Line2" class="form-control" />
  </div>
  <div class="form-group">
    <label>Line 3:</label><input asp-for="Line3" class="form-control" />
  </div>
  <div class="form-group">
    <label>City:</label><input asp-for="City" class="form-control" />
  </div>
  <div class="form-group">
    <label>State:</label><input asp-for="State" class="form-control" />
  </div>
  <div class="form-group">
    <label>Zip:</label><input asp-for="Zip" class="form-control" />
  </div>
  <div class="form-group">
    <label>Country:</label><input asp-for="Country" class="form-control" />
  </div>
  <h3>Options</h3>
  <div class="checkbox">
    <label>
      <input asp-for="GiftWrap" /> Gift wrap these items
    </label>
  </div>
  <div class="text-center">
    <input class="btn btn-primary" type="submit" value="Complete Order" />
  </div>
</form>

```

For each of the properties in the model, I have created a label element and an input element to capture the user input, formatted with Bootstrap. The `asp-for` attribute on the input elements is handled by a built-in tag helper that generates the `type`, `id`, `name`, and `value` attributes based on the specified model property, as described in Chapter 24.

You can see the effect of the new action method and view by starting the application, clicking the cart button at the top of the page, and then clicking the Checkout button, as shown in Figure 10-4. You can also reach this point by requesting the `/Cart/Checkout` URL.



The screenshot shows a web browser window with the URL `localhost:60000/Order/Checkout`. The page header includes the "SPORTS STORE" logo and a cart summary: "Your cart: 1 item(s) \$275.00". On the left, there are navigation buttons for "Home", "Chess", "Soccer", and "Watersports". The main content area is titled "Check out now" and includes the instruction "Please enter your details, and we'll ship your goods right away!". Below this is a "Ship to" section with a "Name:" label and an empty text input field. The "Address" section follows, with labels for "Line 1:", "Line 2:", and "Line 3:", each with an empty text input field. Below the address fields are labels for "City:", "State:", "Zip:", and "Country:", each with an empty text input field. At the bottom of the form is an "Options" section with a checkbox labeled "Gift wrap these items" and a blue "Complete Order" button.

SPORTS STORE Your cart: 1 item(s) \$275.00

Home
Chess
Soccer
Watersports

Check out now

Please enter your details, and we'll ship your goods right away!

Ship to

Name:

Address

Line 1:

Line 2:

Line 3:

City:

State:

Zip:

Country:

Options

Gift wrap these items

[Complete Order](#)

Figure 10-4. The shipping details form

Implementing Order Processing

I will process orders by writing them to the database. Most e-commerce sites would not simply stop there, of course, and I have not provided support for processing credit cards or other forms of payment. But I want to keep things focused on MVC, so a simple database entry will do.

Extending the Database

Adding a new kind of model to the database is simple once the basic plumbing that I created in Chapter 8 is in place. First, I added a new property to the database context class, as shown in Listing 10-13.

Listing 10-13. Adding a Property in the ApplicationDbContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) { }

        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

This change is enough for Entity Framework Core to create a database migration that will allow Order objects to be stored in the database. To create the migration, open a new command prompt or PowerShell window, navigate to the SportsStore project folder (which contains the Startup.cs file), and run the following command:

```
dotnet ef migrations add Orders
```

This command tells Entity Framework Core to take a new snapshot of the application data model, work out how it differs from the previous database version, and generate a new migration called Orders. The new migration will be applied automatically when the application starts because SeedData calls the Migrate method provided by Entity Framework Core.

RESETTING THE DATABASE

When you are making frequent changes to the model, there will come a point when your migrations and your database schema get out of sync. The easiest thing to do is delete the database and start over. However, this applies only during development, of course, because you will lose any data you have stored.

To delete the database, run the following command in the SportsStore project folder:

```
dotnet ef database drop --force
```

Once the database has been removed, run the following command from the SportsStore folder to re-create the database and apply the migrations you have created by running the following command:

```
dotnet ef database update
```

This will reset the database so that it accurately reflects your model and allow you to return to developing your application.

Creating the Order Repository

I am going to follow the same pattern I used for the product repository to provide access to the Order objects. I added a class file called `IOrderRepository.cs` to the Models folder and used it to define the interface shown in Listing 10-14.

Listing 10-14. The Contents of the `IOrderRepository.cs` File in the Models Folder

```
using System.Linq;

namespace SportsStore.Models {

    public interface IOrderRepository {

        IQueryable<Order> Orders { get; }
        void SaveOrder(Order order);
    }
}
```

To implement the order repository interface, I added a class file called `EFOOrderRepository.cs` to the Models folder and defined the class shown in Listing 10-15.

Listing 10-15. The Contents of the `EFOOrderRepository.cs` File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace SportsStore.Models {

    public class EFOOrderRepository : IOrderRepository {
        private ApplicationDbContext context;

        public EFOOrderRepository(ApplicationDbContext ctx) {
            context = ctx;
        }
    }
}
```

```

public IQueryable<Order> Orders => context.Orders
    .Include(o => o.Lines)
    .ThenInclude(l => l.Product);

public void SaveOrder(Order order) {
    context.AttachRange(order.Lines.Select(l => l.Product));
    if (order.OrderID == 0) {
        context.Orders.Add(order);
    }
    context.SaveChanges();
}
}
}

```

This class implements `IOrderRepository` using Entity Framework Core, allowing the set of `Order` objects that have been stored to be retrieved and allowing orders to be created or changed.

UNDERSTANDING THE ORDER REPOSITORY

There is a little extra work required to implement the repository for the orders in Listing 10-15. Entity Framework Core requires instruction to load related data if it spans multiple tables. In the listing, I used the `Include` and `ThenInclude` methods to specify that when an `Order` object is read from the database, the collection associated with the `Lines` property should also be loaded along with each `Product` object associated with each collection object.

```

...
public IQueryable<Order> Orders => context.Orders
    .Include(o => o.Lines)
    .ThenInclude(l => l.Product);
...

```

This ensures that I receive all the data objects that I need without having to perform the queries and assemble the data directly.

An additional step is required when I store an `Order` object in the database. When the user's cart data is deserialized from the session store, the JSON package creates new objects that are not known to Entity Framework Core, which then tries to write all the objects into the database. For the `Product` objects, this means that Entity Framework Core tries to write objects that have already been stored, which causes an error. To avoid this problem, I notify Entity Framework Core that the objects exist and shouldn't be stored in the database unless they are modified, as follows:

```

...
context.AttachRange(order.Lines.Select(l => l.Product));
...

```

This ensures that Entity Framework Core won't try to write the deserialized `Product` objects that are associated with the `Order` object.

In Listing 10-16, I have registered the order repository as a service in the `ConfigureServices` method of the `Startup` class.

Listing 10-16. Registering the Order Repository Service in the `Startup.cs` File in the `SportsStore` Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration["Data:SportStoreProducts:ConnectionString"]));
    services.AddTransient<IProductRepository, EFProductRepository>();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddTransient<IOrderRepository, EFOrderRepository>();
    services.AddMvc();
    services.AddMemoryCache();
    services.AddSession();
}
...
```

Completing the Order Controller

To complete the `OrderController` class, I need to modify the constructor so that it receives the services it requires to process an order, and I need to add a new action method that will handle the HTTP form POST request when the user clicks the Complete Order button. Listing 10-17 shows both changes.

Listing 10-17. Completing the Controller in the `OrderController.cs` File in the `Controllers` Folder

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

namespace SportsStore.Controllers {

    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;

        public OrderController(IOrderRepository repoService, Cart cartService) {
            repository = repoService;
            cart = cartService;
        }

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

        [HttpPost]
        public IActionResult Checkout(Order order) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
        }
    }
}
```

```

        if (ModelState.IsValid) {
            order.Lines = cart.Lines.ToArray();
            repository.SaveOrder(order);
            return RedirectToAction(nameof(Completed));
        } else {
            return View(order);
        }
    }

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

```

The Checkout action method 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 binding system so that I can receive the `Order` object, which I then complete using data from the `Cart` and store in the repository.

MVC checks the validation constraints that I applied to the `Order` class using the data annotation attributes, and any validation problems are passed to the action method through the `ModelState` property. I can see whether there are any problems by checking the `ModelState.IsValid` property. 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 27 and 28.

UNIT TEST: ORDER PROCESSING

To perform unit testing for the `OrderController` class, I need to test the behavior of the POST version of the Checkout method. Although the method looks short and simple, the use of MVC 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, which I defined in a class file called `OrderControllerTests.cs` in the `SportsStore.Tests` project:

```

using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class OrderControllerTests {

        [Fact]
        public void Cannot_Checkout_Empty_Cart() {

```

```

    // Arrange - create a mock repository
    Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
    // Arrange - create an empty cart
    Cart cart = new Cart();
    // Arrange - create the order
    Order order = new Order();
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.Object, cart);

    // Act
    ViewResult result = target.Checkout(order) as ViewResult;

    // Assert - check that the order hasn't been stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
    // Assert - check that the method is returning the default view
    Assert.True(string.IsNullOrEmpty(result.ViewName));
    // Assert - check that I am passing an invalid model to the view
    Assert.False(result.ViewData.ModelState.IsValid);
}
}
}
}
}

```

This test ensures that I cannot check out with an empty cart. I check this by ensuring that `SaveOrder` of the mock `IOrderRepository` implementation is never called, that the view 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):

```

...
[Fact]
public void Cannot_Checkout_Invalid_ShippingDetails() {

    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.Object, cart);
    // Arrange - add an error to the model
    target.ModelState.AddModelError("error", "error");

    // Act - try to checkout
    ViewResult result = target.Checkout(new Order()) as ViewResult;

    // Assert - check that the order hasn't been passed stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
    // Assert - check that the method is returning the default view
    Assert.True(string.IsNullOrEmpty(result.ViewName));
}

```

```

    // Assert - check that I am passing an invalid model to the view
    Assert.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:

```

...
[Fact]
public void Can_Checkout_And_Submit_Order() {
    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.Object, cart);

    // Act - try to checkout
    RedirectToActionResult result =
        target.Checkout(new Order()) as RedirectToActionResult;

    // Assert - check that the order has been stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Once);
    // Assert - check that the method is redirecting to the Completed action
    Assert.Equal("Completed", result.ActionName);
}
...

```

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 `Order` class.

Displaying Validation Errors

MVC will use the validation attributes applied to the `Order` class to validate user data. However, I need to make a simple change to display any problems. This relies on another built-in tag helper that inspects the validation state of the data provided by the user and adds warning messages for each problem that has been discovered. Listing 10-18 shows the addition of an HTML element that will be processed by the tag helper to the `Checkout.cshtml` file.

Listing 10-18. Adding a Validation Summary to the Checkout.cshtml File in the Views/Order Folder

```
@model Order
```

```
<h2>Check out now</h2>
```

```
<p>Please enter your details, and we'll ship your goods right away!</p>
```

```
<div asp-validation-summary="All" class="text-danger"></div>
```

```
<form asp-action="Checkout" method="post">
```

```
  <h3>Ship to</h3>
```

```
  ...
```

With this simple change, validation errors are reported to the user. To see the effect, go to the `/Order/Checkout` URL and try to check out without selecting any products or filling in any shipping details, as shown in Figure 10-5. The tag helper that generates these messages is part of the model validation system, which I describe in detail in Chapter 27.

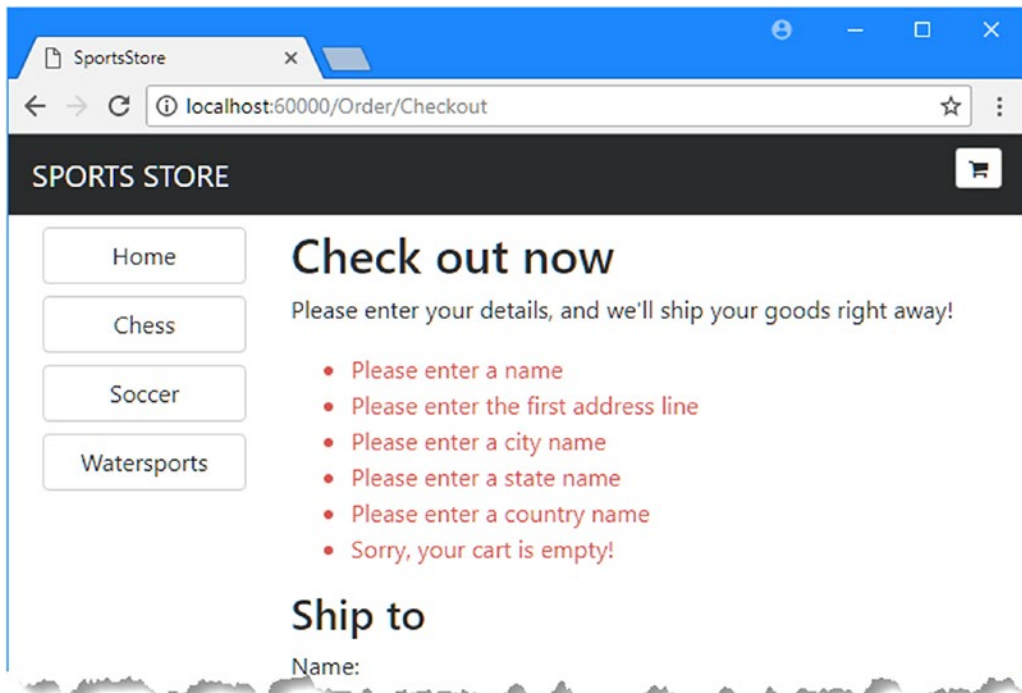


Figure 10-5. Displaying validation messages

■ **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 MVC 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 and processed and the result page has been 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 27.

Displaying a Summary Page

To complete the checkout process, I need to create the view that will be shown when the browser is redirected to the Completed action on the Order controller. I added a Razor view file called Completed.cshtml to the Views/Order folder and added the markup shown in Listing 10-19.

Listing 10-19. The Contents of the Completed.cshtml File in the Views/Order Folder

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

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 Completed action method. 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 10-6.

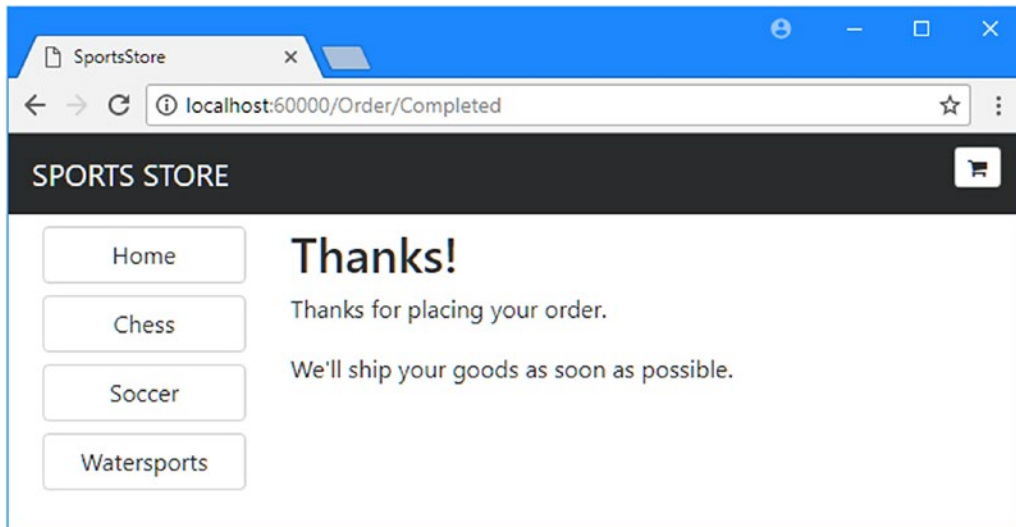


Figure 10-6. The completed order summary view

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 causing problems or inconsistencies elsewhere. For example, I could change the way that orders are stored 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 add the features required to administer the SportsStore application.