TH 05:

Hoàn thành giỏ hàng

5.1 Điều chỉnh Cart Model với Service

Chúng ta đã định nghĩa một lớp Cart model trong phần trước và trình bày cách nó có thể được lưu trữ bằng tính năng phiên (session), cho phép người dùng tạo một danh sách sản phẩm để mua. Trách nhiệm quản lý tính bền vững của lớp Cart thuộc về Cart Razor Page, trang này phải xử lý việc nhận và lưu trữ các đối tượng Cart dưới dạng dữ liệu phiên.

Vấn đề với cách tiếp cận này là sẽ phải sao chép mã lấy và lưu trữ các đối tượng Giỏ hàng (Cart) trong bất kỳ Razor Page hoặc controller nào khác sử dụng chúng. Trong phần này, chúng ta sẽ sử dụng tính năng dịch vụ (services) nằm ở trung tâm của ASP.NET Core để đơn giản hóa cách quản lý các đối tượng Giỏ hàng, giải phóng các thành phần riêng lẻ khỏi phải xử lý trực tiếp các chi tiết.

Các dịch vụ thường được sử dụng để ẩn chi tiết về cách triển khai giao diện khỏi các thành phần phụ thuộc vào chúng. Nhưng các dịch vụ cũng có thể được sử dụng để giải quyết nhiều vấn đề khác và có thể được sử dụng để định hình và định hình lại một ứng dụng, ngay cả khi bạn đang làm việc với các lớp cụ thể như Cart.

5.1.1 Tạo một Storage-Aware Cart Class

Bước đầu tiên trong việc sắp xếp lại cách sử dụng lớp Cart sẽ là tạo một lớp con có khả năng tự lưu trữ bằng trạng thái phiên. Để chuẩn bị, áp dụng từ khóa virtual cho lớp Cart, như trong Listing 5-1, để có thể ghi đè các thành viên.

Listing 5-1. Áp dụng từ khóa Virtual trong file Cart.cs trong thư mục SportsStore/Models

```
namespace SportsStore.Models

{

    public class Cart

    {
```

```
        public List<CartLine> Lines { get; set; } = new
List<CartLine>();

        public virtual void AddItem(Product product, int quantity)

        {

            CartLine? line = Lines

                .Where(p => p.Product.ProductID ==
product.ProductID)

                .FirstOrDefault();

            if (line == null)

            {

                Lines.Add(new CartLine

                {

                    Product = product,

                    Quantity = quantity

                });

            }

            else

            {

                line.Quantity += quantity;

            }

        }

        public virtual void RemoveLine(Product product) =>

           Lines.RemoveAll(l => l.Product.ProductID ==
product.ProductID);

        public decimal ComputeTotalValue() =>

            Lines.Sum(e => e.Product.Price * e.Quantity);

        public virtual void Clear() => Lines.Clear();

    }

    public class CartLine

    {

        public int CartLineID { get; set; }
```

```
        public Product Product { get; set; } = new();

        public int Quantity { get; set; }

    }

}
```

Tiếp theo, thêm một class file có tên SessionCart.cs vào thư mục Models và sử dụng nó để định nghĩa lớp được hiển thị trong Listing 5-2.

Listing 5-2. Nội dung của file SessionCart.cs trong thư mục SportsStore/Models

```
using System.Text.Json.Serialization;

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");
        }
    }
}
```

Lớp SessionCart kế thừa lớp Cart và ghi đè các phương thức AddItem, RemoveLine và Clear để chúng gọi các triển khai cơ sở và sau đó lưu trữ trạng thái cập nhật trong phiên bằng các phương thức mở rộng trên giao diện ISession. Phương thức tĩnh GetCart là một factory để tạo các đối tượng SessionCart và cung cấp cho chúng một đối tượng ISession để chúng có thể tự lưu trữ.

Việc nắm bắt đối tượng ISession hơi phức tạp một chút. Chúng ta nhận được một phiên bản của dịch vụ IHttpContextAccessor, dịch vụ này cung cấp quyền truy cập vào một đối tượng HttpContext, đến lượt nó, cung cấp ISession. Cách tiếp cận gián tiếp này là bắt buộc vì phiên này không được cung cấp dưới dạng dịch vụ thông thường.

5.1.2 Đăng ký dịch vụ (Service)

Bước tiếp theo là tạo một dịch vụ cho lớp Cart. Mục tiêu là đáp ứng các yêu cầu đối với đối tượng Cart bằng đối tượng SessionCart sẽ tự lưu trữ liền mạch. Khai báo đăng ký như trong Listing 5-3.

Listing 5-3. Tạo dịch vụ Giỏ hàng trong file Program.cs trong thư mục SportsStore

```
…


//TH05
builder.Services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
builder.Services.AddSingleton<IHttpContextAccessor,
HttpContextAccessor>();


var app = builder.Build();
…
```

Phương thức AddScoped chỉ định rằng cùng một đối tượng sẽ được sử dụng để đáp ứng các yêu cầu liên quan cho Cart instances. Chúng ta có thể định cấu hình các yêu cầu liên quan như thế nào, nhưng theo mặc định, điều đó có nghĩa là mọi Giỏ hàng được yêu cầu bởi các thành phần xử lý cùng một HTTP request sẽ nhận được cùng một đối tượng.

Thay vì cung cấp cho phương thức AddScoped một ánh xạ kiểu (type mapping), như đã làm với kho lưu trữ, ở đây đã chỉ định một biểu thức lambda sẽ được gọi để đáp ứng các yêu cầu Giỏ hàng. Biểu thức nhận tập hợp các dịch vụ đã được đăng ký và chuyển tập hợp này tới phương thức GetCart của lớp SessionCart. Kết quả là các yêu cầu đối với dịch vụ Cart sẽ được xử lý bằng cách tạo các đối tượng SessionCart, các đối tượng này sẽ tự tuần tự hóa dưới dạng dữ liệu phiên khi chúng được sửa đổi.

Ở đây cũng đã thêm một dịch vụ bằng phương thức AddSingleton, phương thức này chỉ định rằng cùng một đối tượng sẽ luôn được sử dụng. Dịch vụ đã tạo yêu cầu ASP.NET Core sử dụng lớp HttpContextAccessor khi cần triển khai giao diện IHttpContextAccessor. Dịch vụ này là bắt buộc để chúng ta có thể truy cập phiên hiện tại trong lớp SessionCart.

5.1.3 Đơn giản hóa Cart Razor Page

Lợi ích của việc tạo loại dịch vụ này là nó cho phép đơn giản hóa mã nơi sử dụng đối tượng Giỏ hàng. Trong Listing 5-4, đã làm lại page model class cho Cart Razor Page to để tận dụng lợi thế của dịch vụ mới.

Listing 5-4. Sử dụng Cart Service trong Cart.cshtml.cs File trong thư mục SportsStore/Pages

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using SportsStore.Infrastructure;
using SportsStore.Models;
namespace SportsStore.Pages
{
    public class CartModel : PageModel
    {
        private IStoreRepository repository;
        public CartModel(IStoreRepository repo, Cart cartService)
        {
            repository = repo;
            Cart = cartService;
        }
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; } = "/";
        public void OnGet(string returnUrl)
        {
            ReturnUrl = returnUrl ?? "/";
            //Cart = HttpContext.Session.GetJson<Cart>("cart") ??
new Cart();
        }
        public IActionResult OnPost(long productId, string
returnUrl)
        {
            Product? product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            if (product != null)
            {
```

```
                Cart.AddItem(product, 1);
            }
            return RedirectToPage(new { returnUrl = returnUrl });
        }
    }
}
```

Page model class chỉ ra rằng nó cần một đối tượng Cart bằng cách khai báo một đối số hàm tạo, điều này cho phép loại bỏ các câu lệnh tải và lưu trữ phiên khỏi các phương thức xử lý. Kết quả là một page model class đơn giản hơn, tập trung vào vai trò của nó trong ứng dụng mà không phải lo lắng về cách các đối tượng Giỏ hàng được tạo hoặc duy trì. Và vì các dịch vụ có sẵn trong toàn bộ ứng dụng nên bất kỳ thành phần nào cũng có thể chiếm được giỏ hàng của người dùng bằng kỹ thuật tương tự.

UPDATING THE UNIT TESTS

Việc đơn giản hóa lớp CartModel trong Listing 5-4 yêu cầu thay đổi tương ứng đối với các unit tests trong file CartPageTests.cs trong unit test project để Giỏ hàng được cung cấp dưới dạng đối số hàm tạo và không được truy cập thông qua các đối tượng ngữ cảnh. Đây là sự thay đổi trong bài test đọc giỏ hàng:

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Routing;
using Moq;
using SportsStore.Models;
using SportsStore.Pages;
using System.Linq;
using System.Text;
using System.Text.Json;
using Xunit;
namespace SportsStore.Tests
```

```
{
    public class CartPageTests
    {
        [Fact]
        public void Can_Load_Cart()
        {
            // Arrange
            // - create a mock repository
            Product p1 = new Product { ProductID = 1, Name = "P1"
};
            Product p2 = new Product { ProductID = 2, Name = "P2"
};
            Mock<IStoreRepository> mockRepo = new
Mock<IStoreRepository>();
            mockRepo.Setup(m => m.Products).Returns((new Product[]
{
                p1, p2
            }).AsQueryable<Product>());
            // - create a cart
            Cart testCart = new Cart();
            testCart.AddItem(p1, 2);
            testCart.AddItem(p2, 1);
            // Action
            CartModel cartModel = new CartModel(mockRepo.Object,
testCart);
            cartModel.OnGet("myUrl");
            //Assert
            Assert.Equal(2, cartModel.Cart.Lines.Count());
            Assert.Equal("myUrl", cartModel.ReturnUrl);
        }

        [Fact]
```

```
        public void Can_Update_Cart()

        {

            // Arrange

            // - create a mock repository

            Mock<IStoreRepository> mockRepo = new
Mock<IStoreRepository>();

            mockRepo.Setup(m => m.Products).Returns((new Product[]
{

                new Product { ProductID = 1, Name = "P1" }

            }).AsQueryable<Product>());

            Cart testCart = new Cart();

            // Action

            CartModel cartModel = new CartModel(mockRepo.Object,
testCart);

            cartModel.OnPost(1, "myUrl");

            //Assert

            Assert.Single(testCart.Lines);

            Assert.Equal("P1",
testCart.Lines.First().Product.Name);

            Assert.Equal(1, testCart.Lines.First().Quantity);

        }

    }

}
```

Việc sử dụng các dịch vụ sẽ đơn giản hóa quá trình kiểm thử và giúp việc cung cấp cho lớp đang được kiểm thử các phần phụ thuộc của nó trở nên dễ dàng hơn nhiều.

5.2 Hoàn thiện chức năng giỏ hàng

Như vậy đã giới thiệu dịch vụ Giỏ hàng, đã đến lúc hoàn thiện chức năng giỏ hàng bằng cách thêm hai tính năng mới. Đầu tiên sẽ cho phép khách hàng xóa một mặt hàng khỏi giỏ hàng. Tính năng thứ hai sẽ hiển thị tóm tắt giỏ hàng ở đầu trang.

5.2.1 Xóa các mặt hàng khỏi giỏ hàng

Để xóa các mặt hàng khỏi giỏ hàng, cần thêm nút Remove vào nội dung được hiển thị bởi Cart Razor Pages sẽ gửi yêu cầu HTTP POST. Những thay đổi được hiển thị trong Listing 5-5.

Listing 5-5. Xóa các mục trong giỏ hàng trong file Cart.cshtml trong thư mục SportsStore/Pages

```
@page

@model CartModel

<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>

            <th></th>

        </tr>

    </thead>

    <tbody>

        @foreach (var line in Model?.Cart?.Lines ??
Enumerable.Empty<CartLine>())

        {

            <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 class="text-center">
                    <form asp-page-handler="Remove" 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>
```

HTML content mới xác định một dạng HTML. Phương thức xử lý sẽ nhận được yêu cầu được chỉ định bằng thuộc tính asp-page-handler tag helpe, như sau:

```
<form asp-page-handler="Remove" method="post">
```

Nút này yêu cầu một phương thức xử lý mới trong page model class sẽ nhận yêu cầu và sửa đổi giỏ hàng, như trong Listing 5-6.

Listing 5-6. Xóa một mục trong file Cart.cshtml.cs trong thư mục SportsStore/Pages

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


namespace SportsStore.Pages
{
    public class CartModel : PageModel
    {
        private IStoreRepository repository;
        public CartModel(IStoreRepository repo, Cart cartService)
        {
            repository = repo;
            Cart = cartService;
        }
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; } = "/";
        public void OnGet(string returnUrl)
        {
            ReturnUrl = returnUrl ?? "/";
            //Cart = HttpContext.Session.GetJson<Cart>("cart") ??
new Cart();
        }
```

```
        public IActionResult OnPost(long productId, string
returnUrl)
        {
            Product? product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            if (product != null)
            {
                Cart.AddItem(product, 1);
            }
            return RedirectToPage(new { returnUrl = returnUrl });
        }

        public IActionResult OnPostRemove(long productId, string
returnUrl)
        {
            Cart.RemoveLine(Cart.Lines.First(cl =>
                cl.Product.ProductID == productId).Product);
            return RedirectToPage(new { returnUrl = returnUrl });
        }

    }
}
```

Tên được chỉ định có tiền tố On và được cung cấp một hậu tố phù hợp với loại yêu cầu sao cho giá trị Remove sẽ chọn phương thức xử lý OnPostRemove. Phương thức xử lý sử dụng giá trị mà nó nhận được để xác định vị trí mặt hàng trong giỏ hàng và xóa nó.

Khởi động lại ASP.NET Core và yêu cầu http://localhost:5000. Nhấp vào nút Add To Cart để thêm các mặt hàng vào giỏ hàng và sau đó nhấp vào nút Remove. Giỏ hàng sẽ được cập nhật để xóa mặt hàng đã chỉ định, như trong Figure 5-1.

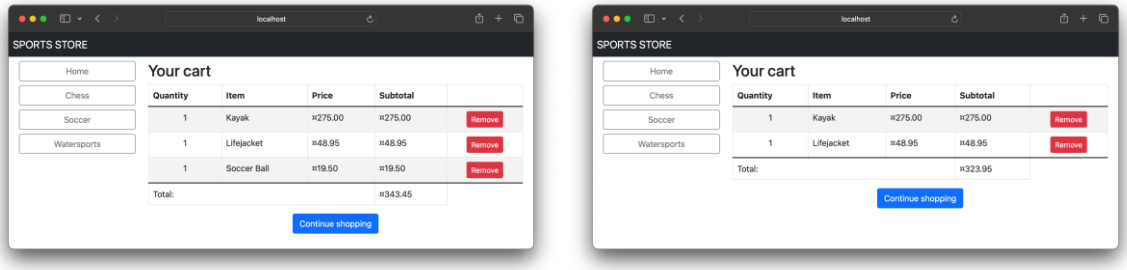


Figure 5-1. Loại bỏ các mặt hàng khỏi giỏ hàng

5.2.2 Thêm Cart Summary Widget

Có thể có một giỏ hàng đang hoạt động nhưng có vấn đề với cách tích hợp nó vào giao diện. Khách hàng chỉ có thể biết trong giỏ hàng của mình có những gì bằng cách xem màn hình tóm tắt giỏ hàng. Và họ chỉ có thể xem màn hình tóm tắt giỏ hàng bằng cách thêm một mặt hàng mới vào giỏ hàng.

Để giải quyết vấn đề này, thêm một tiện ích tóm tắt nội dung của giỏ hàng và có thể nhấp vào tiện ích này để hiển thị nội dung giỏ hàng trong toàn bộ ứng dụng. Để thực hiện việc này giống như cách chúng ta đã thêm tiện ích điều hướng — dưới dạng view component mà chúng ta có thể include vào Razor layout.

5.2.2.1 *Thêm Font Awesome Package*

Là một phần của phần tóm tắt giỏ hàng, chúng ta sẽ hiển thị một nút cho phép người dùng thanh toán. Thay vì hiển thị từ thanh toán trong nút, sẽ sử dụng biểu tượng giỏ hàng. Tiện nhất là sẽ sử dụng gói Font Awesome, đây là một bộ biểu tượng nguồn mở tuyệt vời được tích hợp vào các ứng dụng dưới dạng fonts, trong đó mỗi ký tự trong font là một hình ảnh khác nhau. Có thể tìm hiểu thêm về Font Awesome, bao gồm việc kiểm tra các biểu tượng có trong đó, tại https://fontawesome.com.

Để cài đặt gói client-side package, hãy sử dụng dấu nhắc lệnh PowerShell để chạy lệnh được hiển thị trong Listing 5-7 trong dự án SportsStore.

Listing 5-7. Cài đặt Icon Package

```
libman install font-awesome@5.15.4 -d wwwroot/lib/font-awesome
```

Có thể download font-awesome, đưa vào thư mục wwwroot/lib/font-awesome.

5.2.2.2 *Tạo View Component Class và View*

Thêm một class file có tên CartSummaryViewComponent.cs trong thư mục Components và sử dụng nó để xác định view component được hiển thị trong Listing 5-8.

Listing -8. Nội dung của file CartSummaryViewComponent.cs trong thư mục SportsStore/Components

```
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);
        }
    }
}
```

View component này có thể tận dụng dịch vụ đã tạo trước đó để nhận đối tượng Cart làm đối số hàm tạo. Kết quả là một lớp view component đơn giản truyền Giỏ hàng sang View để tạo đoạn HTML sẽ được đưa vào layout. Để tạo view cho component, tạo thư mục Views/Shared/Components/CartSummary và thêm vào đó một Razor View có tên Default.cshtml với nội dung được hiển thị trong Listing 5-9.

Listing 5-9. Nội dung Default.cshtml thư mục trong Views/Shared/Components/CartSummary

```
@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-
page="/Cart"

       asp-route-
returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">

        <i class="fa fa-shopping-cart"></i>

    </a>

</div>
```

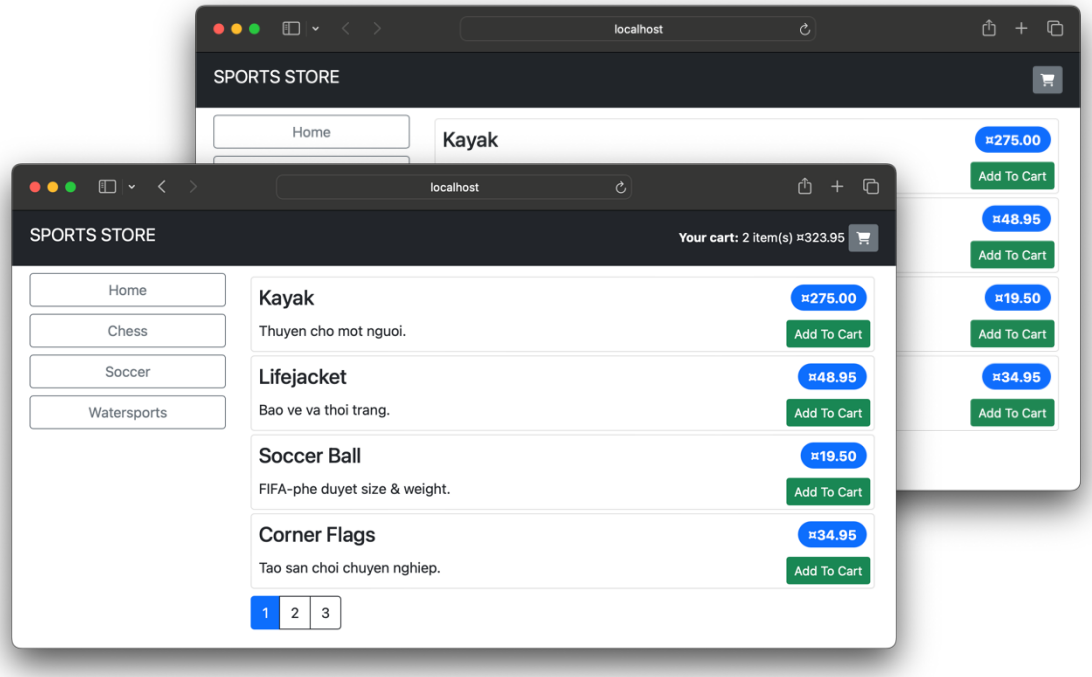
View hiển thị một nút có biểu tượng giỏ hàng Font Awesome và nếu có mặt hàng trong giỏ hàng, view này sẽ cung cấp ảnh chụp nhanh chi tiết số lượng mặt hàng và tổng giá trị của chúng. Bây giờ có một view component và một view, có thể sửa đổi layout sao cho phần tóm tắt giỏ hàng được đưa vào các phản hồi (responses) do Home controller tạo ra, như trong Listing 5-10.

Listing 5-10. Thêm Cart Summary vào file _Layout.cshtml trong thư mục Views/Shared

```
<!DOCTYPE html>

<html>

<head>

    <meta name="viewport" content="width=device-width" />

    <title>SportsStore</title>

    <link href="/lib/bootstrap/css/bootstrap.min.css"
rel="stylesheet" />
```

```
    <link href="/lib/font-awesome/css/all.min.css"
rel="stylesheet" />

</head>

<body>

    <div class="bg-dark text-white p-2">

        <div class="container-fluid">

            <div class="row">

                <div class="col navbar-brand">SPORTS STORE</div>

                <div class="col-6 navbar-text text-end">

                    <vc:cart-summary />

                </div>

            </div>

        </div>

    </div>

    <div class="row m-1 p-1">

        <div id="categories" class="col-3">

            <vc:navigation-menu />

        </div>

        <div class="col-9">

            @RenderBody()

        </div>

    </div>

</body>

</html>
```

Có thể xem tóm tắt giỏ hàng bằng cách khởi động ứng dụng. Khi giỏ hàng trống, chỉ có nút thanh toán được hiển thị. Nếu thêm các mặt hàng vào giỏ hàng thì số lượng mặt hàng và tổng tiền của chúng sẽ được hiển thị, như minh họa trong Figure 5-2. Với sự bổ sung này, khách hàng biết trong giỏ hàng của họ có những gì và có một cách rõ ràng để thanh toán từ cửa hàng.



5.3 Gửi đơn đặt hàng

Tính năng cuối cùng dành cho khách hàng trong SportsStore: khả năng kiểm tra và hoàn tất đơn hàng. Mở rộng data model để cung cấp hỗ trợ thu thập chi tiết giao hàng từ người dùng và thêm hỗ trợ ứng dụng để xử lý các chi tiết đó.

5.3.1 Tạo Model Class

Thêm một class file có tên Order.cs vào thư mục Models và sử dụng nó để định nghĩa lớp được hiển thị trong Listing 5-11. Đây là lớp sẽ sử dụng để thể hiện chi tiết giao hàng cho khách hàng.

Listing 5-11. Nội dung của file Order.cs trong thư mục SportsStore/Models

```
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; } = new
List<CartLine>();

        [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; }

    }

}
```

Sử dụng các thuộc tính validation từ namespace System.ComponentModel.DataAnnotations. Và sử dụng thuộc tính BindNever để ngăn người dùng cung cấp giá trị cho các thuộc tính này trong HTTP request. Đây là một tính năng của hệ thống model binding và nó ngăn ASP.NET Core sử dụng các giá trị từ HTTP request để điền vào các thuộc tính model quan trọng.

5.3.2 Thêm nút Checkout

Mục tiêu là đạt đến điểm mà người dùng có thể nhập chi tiết giao hàng và gửi đơn đặt hàng. Cần thêm nút Checkout vào cart view, như trong Listing 5-12.

Listing 5-12. Thêm nút vào file Cart.cshtml trong thư mục SportsStore/Pages

```
<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>
```

Thay đổi này tạo ra một liên kết kiểu như một nút và khi được nhấp vào, nó sẽ gọi phương thức hành động Checkout của Order controller, sẽ tạo sau. Để cho thấy cách Razor Pages và controllers có thể hoạt động cùng nhau, sẽ xử lý quá trình xử lý đơn hàng trong controller và sau đó quay lại Razor Page khi kết thúc quá trình. Để xem nút Checkout, hãy khởi động lại ASP.NET Core, yêu cầu http:// localhost:5000 và nhấp vào một trong các nút Add To Cart. Nút mới được hiển thị như một phần của bản tóm tắt giỏ hàng, như trong Figure 5-3.

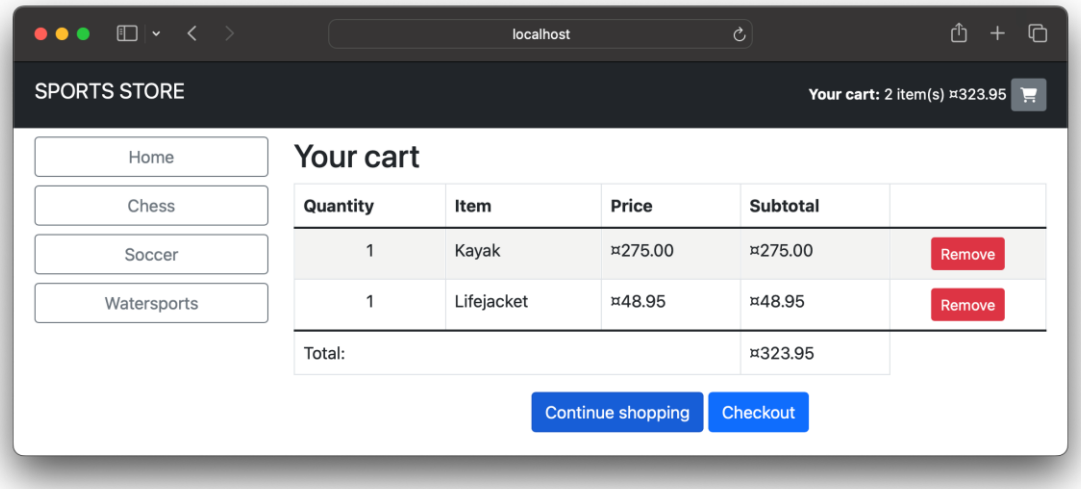


Figure 5-3. Nút Checkout

5.3.3 Tạo Controller và View

Bây giờ cần xác định controller sẽ xử lý đơn hàng. Thêm một class file có tên OrderController.cs vào thư mục Controllers và sử dụng nó để định nghĩa lớp được hiển thị trong Listing 5-13.

Listing 5-13. Nội dung của file OrderController.cs trong thư mục SportsStore/Controllers

```
using Microsoft.AspNetCore.Mvc;

using SportsStore.Models;

namespace SportsStore.Controllers {
```

```
    public class OrderController : Controller
    {
        public ViewResult Checkout() => View(new Order());
    }
}
```

Phương thức Checkout trả về view mặc định và chuyển đối tượng Order mới làm view model. Để tạo view, tạo thư mục Views/Order và thêm vào đó một Razor View có tên là Checkout.cshtml với nội dung được hiển thị trong Listing 5-14.

Listing 5-14. Nội dung của file Checkout.cshtml trong thư mục SportsStore/Views/Order

```
@model Order

<h2>Thanh toan</h2>

<p>Vui lòng nhập thông tin giao hàng<!/p>

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

    <h3>Name</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" /> Gói quà!

        </label>

    </div>

    <div class="text-center">

        <input class="btn btn-primary" type="submit"
value="Complete Order" />

    </div>

</form>
```

Đối với mỗi thuộc tính trong model, tạo nhãn và các phần tử input để ghi lại thông tin đầu vào của người dùng, được tạo kiểu bằng Bootstrap và được định cấu hình bằng tag helper. Thuộc tính asp-for trên các phần tử input được xử lý bởi built-in tag helper để tạo các thuộc tính type, id, name và giá trị dựa trên thuộc tính model đã chỉ định.

Có thể xem form, được hiển thị trong Figure 5-4, bằng cách khởi động lại ASP.NET Core, yêu cầu http:// localhost:5000, thêm một mục vào giỏ và nhấp vào nút Checkout. Hoặc trực tiếp, có thể yêu cầu http://localhost:5000/order/checkout.

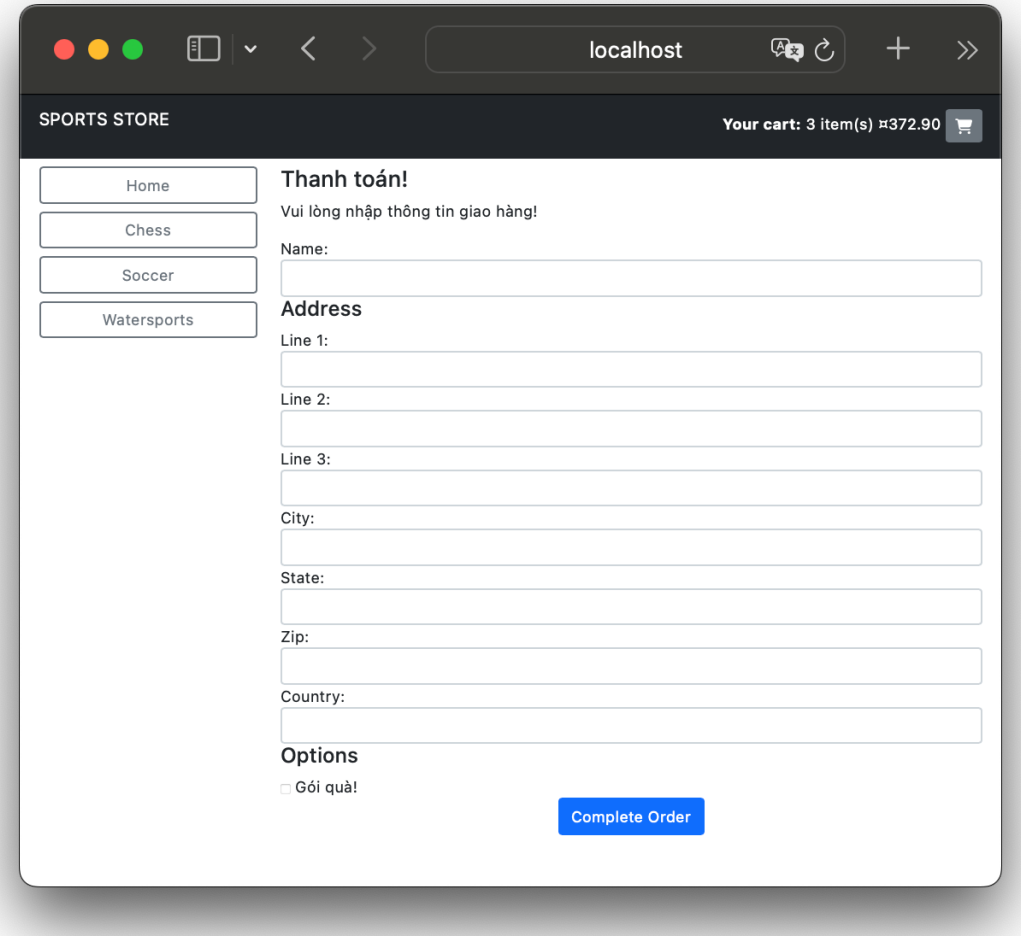


Figure 5-4. Chi tiết form giao hàng

5.3.4 Thực hiện xử lý đơn hàng

Xử lý các đơn đặt hàng bằng cách ghi chúng vào cơ sở dữ liệu. Tất nhiên, hầu hết các trang thương mại điện tử sẽ không chỉ dừng lại ở đó và ở đây chưa cung cấp hỗ trợ xử lý thẻ tín dụng hoặc các hình thức thanh toán khác.

5.3.5 Mở rộng database

Việc thêm một loại model mới vào cơ sở dữ liệu rất đơn giản vì thiết lập ban đầu đã thực hiện trong TH01. Đầu tiên, thêm một thuộc tính mới vào ldatabase context, như trong Listing 5-15.

Listing 5-15. Thêm thuộc tính vào file StoreDbContext.cs trong thư mục SportsStore/Models

```
using System;

using Microsoft.EntityFrameworkCore;



namespace SportsStore.Models

{

    public class StoreDbContext : DbContext

    {

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

        public DbSet<Product> Products => Set<Product>();

        public DbSet<Order> Orders => Set<Order>();

    }

}
```

Thay đổi này đủ để Entity Framework Core tạo ra một database migration cho phép các đối tượng Order được lưu trữ trong cơ sở dữ liệu. Để tạo q migration, hãy sử dụng dấu nhắc lệnh PowerShell để chạy lệnh được hiển thị trong Listing 5-16 trong thư mục SportsStore.

Listing 5-16. Tạo Migration

```
dotnet ef migrations add Orders
```

Lệnh này yêu cầu Entity Framework Core chụp snapshot mới của data model ứng dụng, tìm hiểu xem nó khác với phiên bản cơ sở dữ liệu trước đó như thế nào và tạo một migration mới có tên là Orders. Migration mới sẽ được áp dụng tự động khi ứng dụng khởi động vì SeedData gọi phương thức Migrate do Entity Framework Core cung cấp.

THIẾT LẬP LẠI CƠ SỞ DỮ LIỆU

Khi chúng ta thường xuyên thực hiện các thay đổi đối với model, sẽ có lúc migrations và lược đồ cơ sở dữ liệu không đồng bộ. Cách dễ nhất để làm là xóa cơ sở dữ liệu và bắt đầu lại. Tuy nhiên, điều này chỉ nên áp dụng trong quá trình phát triển vì sẽ mất mọi dữ liệu đã lưu trữ. Chạy lệnh sau để xóa cơ sở dữ liệu:

```
dotnet ef database drop --force --context StoreDbContext
```

Khi cơ sở dữ liệu đã bị xóa, hãy chạy lệnh sau từ thư mục SportsStore để tạo lại cơ sở dữ liệu và áp dụng các migrations đã tạo:

```
dotnet ef database update --context StoreDbContext
```

Việc migrations cũng sẽ được lớp SeedData áp dụng nếu mới khởi động ứng dụng. Dù bằng cách nào, cơ sở dữ liệu sẽ được đặt lại để phản ánh chính xác mô hình dữ liệu và cho phép quay lại phát triển ứng dụng.

5.3.6 Tạo Order Repository

Sẽ làm các đã sử dụng cho kho sản phẩm để cung cấp quyền truy cập vào các đối tượng Order. Thêm một class file có tên IOrderRepository.cs vào thư mục Models và sử dụng nó để xác định interface được hiển thị trong Listing 5-17.

Listing 5-17. Nội dung của file IOrderRepository.cs trong thư mục SportsStore/Models

```
namespace SportsStore.Models

{

    public interface IOrderRepository

    {

        IQueryable<Order> Orders { get; }

        void SaveOrder(Order order);

    }

}
```

Để triển khai order repository interface, thêm một class file có tên EFOrderRepository.cs vào thư mục Models và định nghĩa lớp được hiển thị trong Listing 5-18.

Listing 5-18. Nội dung của file EFOrderRepository.cs trong thư mục SportsStore/Models

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models

{

    public class EFOrderRepository : IOrderRepository
```

```
    {
        private StoreDbContext context;
        public EFOrderRepository(StoreDbContext 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();
        }
    }
}
```

Lớp này triển khai giao diện IOrderRepository bằng cách sử dụng Entity Framework Core, cho phép truy xuất tập hợp các đối tượng Order đã được lưu trữ và cho phép tạo hoặc thay đổi các đơn hàng.

ORDER REPOSITORY

Entity Framework Core yêu cầu hướng dẫn tải dữ liệu liên quan nếu nó trải rộng trên nhiều bảng. Trong Listing 5-18, đã sử dụng các phương thức Include và ThenInclude để chỉ định rằng khi một đối tượng Order được đọc từ cơ sở dữ liệu, thì collection được liên kết với thuộc tính Lines cũng phải được tải cùng với từng đối tượng Product được liên kết với từng đối tượng collection.

```
public IQueryable<Order> Orders => context.Orders

            .Include(o => o.Lines)

            .ThenInclude(l => l.Product);
```

Điều này đảm bảo rằng nhận được tất cả các đối tượng dữ liệu cần mà không cần phải thực hiện các truy vấn riêng biệt rồi tự tập hợp dữ liệu.

Một bước bổ sung cũng được yêu cầu khi lưu trữ đối tượng Order trong cơ sở dữ liệu. Khi dữ liệu giỏ hàng của người dùng được de-serialized khỏi session store, các đối tượng mới sẽ được tạo mà Entity Framework Core chưa biết đến, sau đó thực thể này sẽ cố gắng ghi tất cả các đối tượng vào cơ sở dữ liệu. Đối với các đối tượng Product được liên kết với Order, điều này có nghĩa là Entity Framework Core cố gắng ghi các đối tượng đã được lưu trữ, điều này gây ra lỗi. Để tránh sự cố này, thông báo cho Entity Framework Core rằng các đối tượng tồn tại và không nên được lưu trữ trong cơ sở dữ liệu trừ khi chúng được sửa đổi, như sau:

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

Điều này đảm bảo rằng Entity Framework Core sẽ không cố gắng ghi các đối tượng Product đã de-serialized được liên kết với đối tượng Order.

Trong Listing 5-19, đăng ký order repository như một service trong phương thức configureServices của file Program.cs.

Listing 5-19. Đăng ký Order Repository Service trong Program.cs File trong thư mục SportsStore

```
builder.Services.AddScoped<IStoreRepository, EFStoreRepository>();

//TH05

builder.Services.AddScoped<IOrderRepository, EFOrderRepository>();
```

5.3.7 Hoàn chỉnh Order Controller

Để hoàn thành lớp OrderController, cần sửa đổi hàm tạo để nó nhận các dịch vụ cần thiết để xử lý đơn hàng và thêm một phương thức hành động sẽ xử lý POST request của HTTP form khi người dùng nhấp vào nút Complete Order. Listing 5-20 cho thấy cả hai thay đổi.

Listing 5-20. Hoàn chỉnh Controller trong file OrderController.cs trong thư mục SportsStore/Controllers

```
using Microsoft.AspNetCore.Mvc;

using SportsStore.Models;

namespace SportsStore.Controllers

{

    public class OrderController : Controller

    {

        private IOrderRepository repository;

        private Cart cart;

        public OrderController(IOrderRepository repoService, Cart
cartService)

        {

            repository = repoService;

            cart = cartService;

        }

        public ViewResult 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);

                cart.Clear();
```

```
                return RedirectToPage("/Completed", new { orderId
= order.OrderID });

            }

            else

            {

                return View();

            }

        }

    }

}
```

Phương thức hành động Checkout được chỉ định bằng thuộc tính HttpPost, có nghĩa là nó sẽ được sử dụng để xử lý các POST requests — trong trường hợp này là khi người dùng gửi form.

Trong TH02, chúng ta dã sử dụng tính năng model binding trong ASP.NET Core để nhận các giá trị dữ liệu đơn giản từ request. Tính năng tương tự này được sử dụng trong phương thức hành động mới để nhận đối tượng Order đã hoàn thành. Khi một yêu cầu được xử lý, hệ thống model binding sẽ cố gắng tìm các giá trị cho các thuộc tính được xác định bởi lớp Order. Điều này hoạt động trên cơ sở nỗ lực tối đa, có nghĩa là có thể nhận được một đối tượng Order thiếu giá trị thuộc tính nếu không có data item tương ứng trong request.

Để đảm bảo có được dữ liệu yêu cầu, áp dụng các thuộc tính validation cho lớp Order. ASP.NET Core kiểm tra các ràng buộc validation đã áp dụng cho lớp Order và cung cấp chi tiết về kết quả thông qua thuộc tính ModelState. Có thể xem liệu có vấn đề gì không bằng cách kiểm tra thuộc tính ModelState.IsValid. Gọi phương thức ModelState.AddModelError để đăng ký thông báo lỗi nếu không có mặt hàng nào trong giỏ hàng.

UNIT TEST: ORDER PROCESSING

Để thực hiện kiểm tra đơn vị cho lớp OrderController, cần kiểm tra hoạt động của phiên bản POST của phương thức Checkout. Mặc dù phương pháp này trông ngắn gọn và đơn giản, nhưng việc sử dụng model binding có nghĩa là có rất nhiều điều đang diễn ra ở phía sau cần được thử nghiệm.

Chỉ muốn xử lý đơn đặt hàng nếu có mặt hàng trong giỏ hàng và khách hàng đã cung cấp chi tiết giao hàng hợp lệ. Trong tất cả các trường hợp khác, khách hàng sẽ thấy có lỗi. Đây là phương thức test đầu tiên đã định nghĩa trong class file có tên OrderControllerTests.cs trong SportsStore.Tests. 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);

        }

    }

}
```

Test này đảm bảo rằng không thể thanh toán khi giỏ hàng trống. Kiểm tra điều này bằng cách đảm bảo rằng SaveOrder của việc triển khai mock IOrderRepository không bao giờ được gọi, rằng view mà phương thức trả về là view mặc định (sẽ hiển thị lại dữ liệu được khách hàng nhập và cho họ cơ hội sửa nó) và model state được chuyển tới view đã được đánh dấu là không hợp lệ.

Điều này có vẻ giống như một tập hợp các khẳng định mang tính ràng buộc, nhưng cần cả ba điều này để chắc chắn rằng mình có hành vi đúng đắn. Phương thức test tiếp theo hoạt động theo cách tương tự nhưng đưa lỗi vào view model để mô phỏng sự cố do model binder báo cáo (điều này sẽ xảy ra trong quá trình xử ký khi khách hàng nhập dữ liệu giao hàng không hợp lệ):

```
        [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);

        }
```

Đã xác định rằng giỏ hàng trống hoặc chi tiết không hợp lệ sẽ ngăn việc xử lý đơn hàng, cần đảm bảo rằng xử lý đơn hàng khi thích hợp. Đây là 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
```

```
            RedirectToPageResult? result =

                    target.Checkout(new Order()) as
RedirectToPageResult;

            // 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?.PageName);

        }
```

Không cần kiểm tra xem có thể xác định chi tiết giao hàng hợp lệ hay không. việc này được model binder xử lý tự động bằng cách sử dụng các thuộc tính được áp dụng cho các thuộc tính của lớp Order.

5.3.8 Hiển thị Validation Errors

ASP.NET Core sử dụng các thuộc tính validation được áp dụng cho lớp Order để xác thực dữ liệu người dùng, nhưng cần thực hiện một thay đổi đơn giản để hiển thị mọi vấn đề. Điều này dựa vào một built-in tag helper khác để kiểm tra validation state của dữ liệu do người dùng cung cấp và thêm thông báo cảnh báo cho từng vấn đề đã được phát hiện. Listing 5-21 cho thấy việc bổ sung một phần tử HTML sẽ được tag helper xử lý vào file Checkout.cshtml.

Listing 5-21. Thêm Validation Summary vào file Checkout.cshtml trong thư mục SportsStore/Views/Order

```
<p>Vui lòng nhập thông tin giao hàng!</p>

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

Với thay đổi này, validation errors sẽ được thông báo cho người dùng. Để xem, khởi động lại ASP.NET Core, yêu cầu http://localhost:5000/Order/Checkout và nhấp vào nút Complete Order mà không cần điền vào form. ASP.NET Core sẽ xử lý form data, phát hiện rằng không tìm thấy các giá trị được yêu cầu và tạo ra các validation errors được hiển thị trong Figure 5-5.

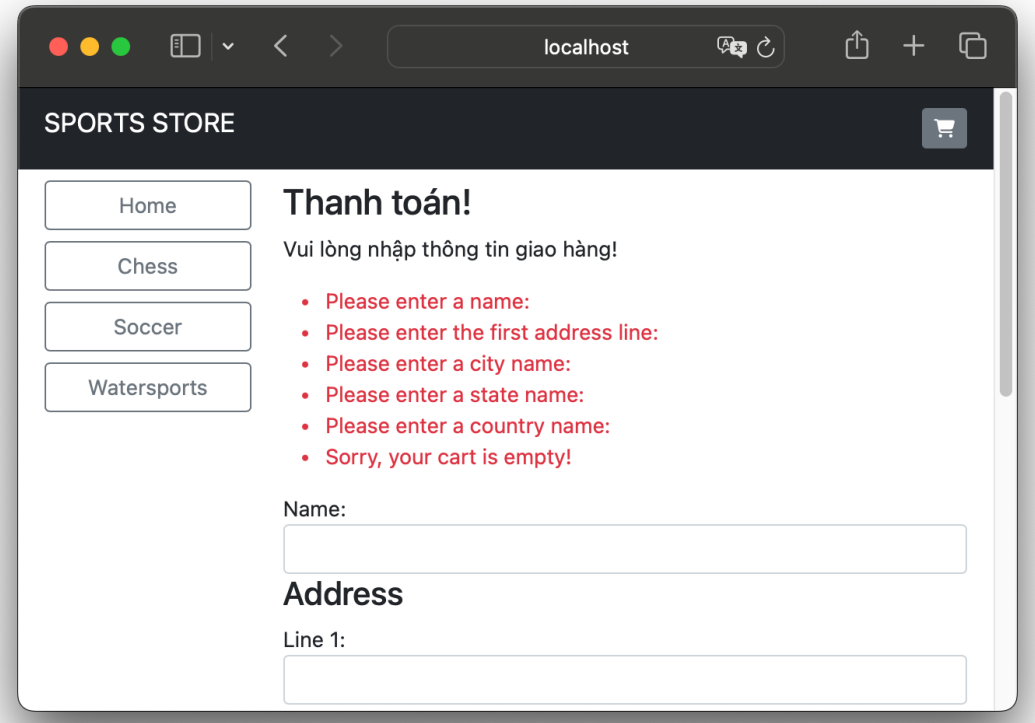


Figure 5-5. Hiển thị thông báo validation

Dữ liệu do người dùng gửi sẽ được gửi đến máy chủ trước khi nó được xác thực, được gọi là xác thực phía máy chủ (server- side validation) và ASP.NET Core có sự hỗ trợ cho việc này. Vấn đề với xác thực phía máy chủ là người dùng không được thông báo về lỗi cho đến khi dữ liệu được gửi đến máy chủ và được xử lý cũng như trang kết quả đã được tạo — việc này có thể mất vài giây trên một máy chủ bận. Vì lý do này, việc xác thực phía máy chủ thường được bổ sung bằng xác thực phía máy khách (client-side validation), trong đó JavaScript được sử dụng để kiểm tra các giá trị mà người dùng đã nhập trước khi form data được gửi đến máy chủ.

5.3.9 Hiển thị Summary Page

Để hoàn tất quy trình thanh toán, tạo Razor Page hiển thị thông báo cảm ơn kèm theo bản tóm tắt đơn đặt hàng. Thêm một Razor Page có tên Completed.cshtml vào thư mục Pages với nội dung được hiển thị trong Listing 5-22.

Listing 5-22. Nội dung của file Completed.cshtml trong thư mục SportsStore/Pages

```
@page

<div class="text-center">

    <h2>Thanks!</h2>

    <p>Cám ơn vì đã đặt  #@OrderId</p>
```

```
    <p>Đơn hàng sẽ được giao trong thời gian sớm nhất!</p>

    <a class="btn btn-primary" asp-controller="Home">Return to
Store</a>

</div>

@functions {

    [BindProperty(SupportsGet = true)]

    public string? OrderId { get; set; }

}
```

Mặc dù Razor Pages thường có các lớp page model nhưng chúng không phải là yêu cầu bắt buộc và các tính năng đơn giản có thể được phát triển mà không cần đến chúng. Trong ví dụ này, đã định nghĩa một thuộc tính có tên OrderId và chỉ định nó bằng thuộc tính BindProperty, thuộc tính này chỉ định rằng giá trị cho thuộc tính này phải được lấy từ yêu cầu của model binding system.

Giờ đây, khách hàng có thể thực hiện toàn bộ quá trình, từ chọn sản phẩm đến thanh toán. Nếu họ cung cấp chi tiết giao hàng hợp lệ (và có mặt hàng trong giỏ hàng), họ sẽ thấy trang tóm tắt khi nhấp vào nút Hoàn tất đơn Complete Order, như trong Figure 5-6.

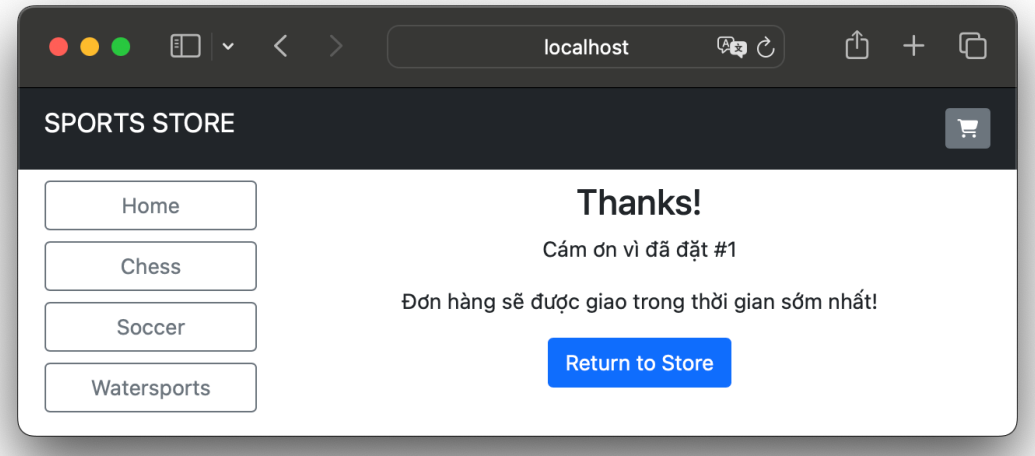


Figure 5-6. Hoàn thành đặt hàng