ГЛАВА 9

SportsStore: навигация

В настоящей главе мы продолжим построение примера приложения SportsStore, добавив к нему поддержку навигации по приложению и начав создавать корзину для покупок.

Добавление навигационных элементов управления

Приложение SportsStore станет более удобным, если пользователи получат возможность навигации среди товаров по категории. Мы будем решать задачу в три этапа.

- Расширим метод действия `List()` в классе `ProductController`, чтобы он был способен фильтровать объекты `Product` в хранилище.
- Пересмотрим и расширим схему URL.
- Создадим список категорий, который будет находиться в боковой панели сайта, подсвечивая текущую категорию и предоставляя ссылки на остальные категории.

Фильтрация списка товаров

Мы начнем с расширения класса модели представления `ProductsListViewModel`, который был добавлен в проект `SportsStore` в предыдущей главе. Нам нужно обеспечить взаимодействие текущей категории с представлением, чтобы визуализировать боковую панель, и это хорошая отправная точка. В листинге 9.1 показаны изменения, внесенные в файл `ProductsListViewModel.cs` из папки `Models/ViewModels`.

Листинг 9.1. Добавление свойства в файле `ProductsListViewModel.cs` из папки `Models/ViewModels`

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

  public class ProductsListViewModel {
    public IEnumerable<Product> Products { get; set; }
    public PagingInfo PagingInfo { get; set; }
    public string CurrentCategory { get; set; }
  }
}
```

В класс ProductsListViewModel добавлено свойство по имени CurrentCategory. Следующий шаг заключается в обновлении класса ProductController, чтобы метод действия List() фильтровал объекты Product по категории и использовал только что добавленное в модель представления свойство для указания категории, выбранной в текущий момент. Изменения приведены в листинге 9.2.

Листинг 9.2. Добавление поддержки категорий к методу действия List() в файле ProductController.cs из папки Controllers

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

namespace SportsStore.Controllers {

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

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

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

В метод действия List() внесены три изменения. Первое — добавлен новый параметр по имени category. Он применяется вторым изменением, которое связано с расширением запроса LINQ. Если значение category не равно null, тогда выбираются только объекты Product с соответствующим значением в свойстве Category. Последнее, третье, изменение касается установки значения свойства CurrentCategory, которое было добавлено в класс ProductsListViewModel. Однако в результате таких изменений значение PagingInfo.TotalItems вычисляется некорректно, потому что оно не принимает во внимание фильтр по категории. Со временем мы все исправим.

Модульное тестирование: обновление существующих модульных тестов

Мы изменили сигнатуру метода действия `List()`, поэтому некоторые существующие методы модульного тестирования перестали компилироваться. Чтобы решить возникшую проблему, в модульных тестах, которые работают с контроллером, методу действия `List()` понадобится передавать в первом параметре значение `null`. Например, раздел действия тестового метода `Can_Paginate()` в файле `ProductControllerTests.cs` должен выглядеть следующим образом:

```
...
[Fact]
public void Can_Paginate() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());

  ProductController controller = new ProductController(mock.Object);
  controller.PageSize = 3;

  // Действие
  ProductsListViewModel result =
    controller.List(null, 2).ViewData.Model as ProductsListViewModel;

  // Утверждение
  Product[] prodArray = result.Products.ToArray();
  Assert.True(prodArray.Length == 2);
  Assert.Equal("P4", prodArray[0].Name);
  Assert.Equal("P5", prodArray[1].Name);
}
...
```

Указывая `null` для `category`, мы получаем все объекты `Product`, которые контроллер извлекает из хранилища, что воспроизводит ситуацию перед добавлением нового параметра. Такого же рода изменение необходимо внести в тестовый метод `Can_Send_Pagination_View_Model()`:

```
...
[Fact]
public void Can_Send_Pagination_View_Model() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
    new Product {ProductID = 4, Name = "P4"},
    new Product {ProductID = 5, Name = "P5"}
  }).AsQueryable<Product>());
  // Организация
  ProductController controller =
    new ProductController(mock.Object) { PageSize = 3 };
```

```
    // Действие
    ProductsListViewModel result =
      controller.List(null, 2).ViewData.Model as ProductsListViewModel;

    // Утверждение
    PagingInfo pageInfo = result.PagingInfo;
    Assert.Equal(2, pageInfo.CurrentPage);
    Assert.Equal(3, pageInfo.ItemsPerPage);
    Assert.Equal(5, pageInfo.TotalItems);
    Assert.Equal(2, pageInfo.TotalPages);
}
...
```

Когда вы примете образ мышления, ориентированный на тестирование, поддержание модульных тестов в синхронизированном состоянии с изменениями кода быстро станет вашей второй натурой.

Чтобы увидеть результат фильтрации по категории, запустим приложение и выберем категорию с помощью показанной ниже строки запроса (замените номер порта тем, который был назначен вашему проекту средой Visual Studio), позаботившись о том, что `Soccer` начинается с прописной буквы `S`:

```
http://localhost:60000/?category=Soccer
```

Отобразятся только товары из категории `Soccer` (рис. 9.1).

Вряд ли пользователи захотят переходить по категориям с применением URL, но здесь было показано, что незначительные изменения в приложении MVC могут оказывать крупное влияние, если базовая структура на месте.

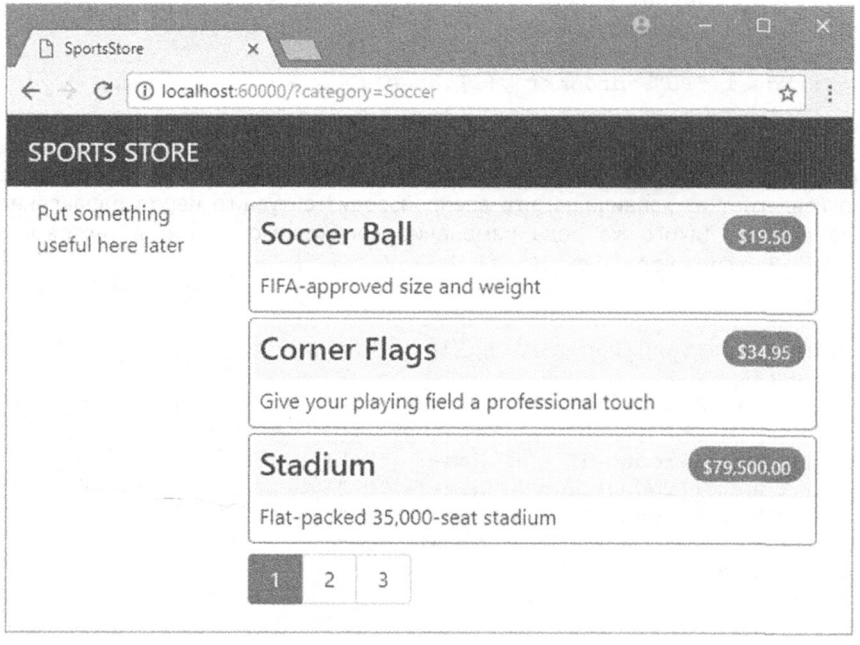

Рис. 9.1. Использование строки запроса для фильтрации по категории

Модульное тестирование: фильтрация по категории

Нам необходим модульный тест для проверки функциональности фильтрации по категории, чтобы удостовериться в том, что фильтр может корректно генерировать сведения о товарах указанной категории. Ниже приведен тестовый метод, добавленный в класс `ProductControllerTests`:

```
...
[Fact]
public void Can_Filter_Products() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
    new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
    new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
    new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
    new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
  }).AsQueryable<Product>());

  // Организация - создание контроллера и установка размера страницы
  // в три элемента
  ProductController controller = new ProductController(mock.Object);
  controller.PageSize = 3;

  // Действие
  Product[] result =
    (controller.List("Cat2", 1).ViewData.Model as ProductsListViewModel)
      .Products.ToArray();

  // Утверждение
  Assert.Equal(2, result.Length);
  Assert.True(result[0].Name == "P2" && result[0].Category == "Cat2");
  Assert.True(result[1].Name == "P4" && result[1].Category == "Cat2");
}
...
```

Тест создает имитированное хранилище, содержащее объекты `Product`, которые относятся к диапазону категорий. С использованием метода действия `List()` запрашивается одна специфическая категория, и результаты проверяются на предмет наличия корректных объектов в правильном порядке.

Улучшение схемы URL

Мало кому понравится видеть либо иметь дело с неуклюжими URL вроде `/?category=Soccer`. Для решения этой проблемы мы намерены изменить схему маршрутизации в методе `Configure()` класса `Startup`, чтобы создать более удобный набор URL (листинг 9.3).

Внимание! Новые маршруты в листинге 9.3 важно добавлять в показанном порядке. Маршруты применяются в порядке, в котором они определены, поэтому изменение порядка может привести к нежелательным эффектам.

Листинг 9.3. Изменение схемы маршрутизации в файле `Startup.cs` из папки `SportsStore`

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.UseDeveloperExceptionPage();
  app.UseStatusCodePages();
  app.UseStaticFiles();
  app.UseMvc(routes => {

    routes.MapRoute(
      name: null,
      template: "{category}/Page{productPage:int}",
      defaults: new { controller = "Product", action = "List" }
    );

    routes.MapRoute(
      name: null,
      template: "Page{productPage:int}",
      defaults: new { controller = "Product",
      action = "List", productPage = 1 }
    );

    routes.MapRoute(
      name: null,
      template: "{category}",
      defaults: new { controller = "Product",
      action = "List", productPage = 1 }
    );

    routes.MapRoute(
      name: null,
      template: "",
      defaults: new { controller = "Product", action = "List",
        productPage = 1 });

    routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
  });
  SeedData.EnsurePopulated(app);
}
...
```

В табл. 9.1 описана схема URL, которую представляют определенные маршруты. Система маршрутизации подробно объясняется в главах 15 и 16.

Таблица 9.1. Сводка по маршрутам

URL	Что делает
`/`	Выводит первую страницу списка товаров всех категорий
`/Page2`	Выводит указанную страницу (в данном случае страницу 2), отображая товары всех категорий
`/Soccer`	Выводит первую страницу товаров указанной категории (`Soccer`)
`/Soccer/Page2`	Выводит указанную страницу (страницу 2) товаров заданной категории (`Soccer`)

Система маршрутизации ASP.NET используется инфраструктурой MVC для обработки входящих запросов от пользователей, но также генерирует исходящие URL, которые соответствуют схеме URL и потому могут встраиваться в веб-страницы. Применение системы маршрутизации для обработки входящих запросов и генерации исходящих URL позволяет гарантировать согласованность всех URL в приложении.

Интерфейс `IUrlHelper` предоставляет доступ к функциональности генерации URL. Мы использовали этот интерфейс и определяемый им метод `Action()` в классе вспомогательной функции дескриптора, созданном в предыдущей главе. Теперь, когда нужно генерировать более сложные URL, необходим способ получения дополнительной информации от представления, не добавляя дополнительные свойства к классу вспомогательной функции дескриптора. К счастью, классы вспомогательных функций дескрипторов обладают удобным средством, которое позволяет получать в одной коллекции все свойства с общим префиксом (листинг 9.4).

Листинг 9.4. Получение значений атрибутов, снабженных префиксом, в файле `PageLinkTagHelper.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
using System.Collections.Generic;

namespace SportsStore.Infrastructure {

  [HtmlTargetElement("div", Attributes = "page-model")]
  public class PageLinkTagHelper : TagHelper {
    private IUrlHelperFactory urlHelperFactory;

    public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
      urlHelperFactory = helperFactory;
    }
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }
    public PagingInfo PageModel { get; set; }
    public string PageAction { get; set; }
    [HtmlAttributeName(DictionaryAttributePrefix = "page-url-")]
    public Dictionary<string, object> PageUrlValues { get; set; }
      = new Dictionary<string, object>();
    public bool PageClassesEnabled { get; set; } = false;
    public string PageClass { get; set; }
    public string PageClassNormal { get; set; }
    public string PageClassSelected { get; set; }

    public override void Process(TagHelperContext context,
          TagHelperOutput output) {
      IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
      TagBuilder result = new TagBuilder("div");
      for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        PageUrlValues["productPage"] = i;
        tag.Attributes["href"] = urlHelper.Action(PageAction, PageUrlValues);
```

```
        if (PageClassesEnabled) {
          tag.AddCssClass(PageClass);
          tag.AddCssClass(i == PageModel.CurrentPage
            ? PageClassSelected : PageClassNormal);
        }
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
      }
      output.Content.AppendHtml(result.InnerHtml);
    }
  }
}
```

Декорирование свойства в классе вспомогательной функции дескриптора посредством атрибута `HtmlAttributeName` позволяет указывать префикс для имен атрибутов элемента, которым в данном случае будет `page-url-`. Значение любого атрибута, чье имя начинается с такого префикса, будет добавлено в словарь, присваиваемый свойству `PageUrlValues`, которое затем передается методу `IUrlHelper.Action()`, чтобы сгенерировать URL для атрибута `href` элементов `a`, выпускаемых вспомогательной функцией дескриптора.

В листинге 9.5 к элементу `div` добавлен новый атрибут, который обрабатывается вспомогательной функцией дескриптора и указывает категорию, применяемую для генерации URL. В представление был добавлен только один атрибут, но в словарь добавился бы любой атрибут с тем же самым префиксом.

Листинг 9.5. Добавление нового атрибута в файле `List.cshtml` из папки `Views/Home`

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  @Html.Partial("ProductSummary", p)
}
<div page-model="@Model.PagingInfo" page-action="List"
                  page-classes-enabled="true"
  page-class="btn" page-class-normal="btn-secondary"
  page-class-selected="btn-primary" page-url-category="@Model.CurrentCategory"
     class="btn-group pull-right m-1">
</div>
```

До внесения этого изменения ссылки для перехода на страницы генерировались так:

```
http://<сервер>:<порт>/Page1
```

Если пользователь щелкнет на страничной ссылке подобного типа, то фильтр по категории утрачивается и приложение отобразит страницу, содержащую товары всех категорий. За счет добавления текущей категории, получаемой из модели представления, взамен генерируются URL следующего вида:

```
http://<сервер>:<порт>/Chess/Page1
```

Когда пользователь щелкает на ссылке такого рода, текущая категория передается методу действия `List()` и фильтрация сохраняется. После внесения данного изменения можно посещать URL вроде `/Chess` или `/Soccer` и наблюдать, что страничные ссылки, расположенные внизу, корректно включают категорию.

Построение меню навигации по категориям

Нам нужно предложить пользователям способ выбора категории, который не предусматривает ввод URL, что означает необходимость в отображении списка доступных категорий с отмеченной текущей категорией, если она есть. По мере построения приложения список категорий будет задействован в нескольких контроллерах, а потому он должен быть самодостаточным и многократно используемым.

В инфраструктуре ASP.NET Core MVC поддерживается концепция *компонентов представлений*, которые идеально подходят для создания единиц вроде многократно используемого навигационного элемента управления. Компонент представления — это класс C#, который предлагает небольшой объем многократно используемой прикладной логики с возможностью выбора и отображения частичных представлений Razor. Компоненты представлений подробно рассматриваются в главе 22.

В данном случае мы создадим компонент представления, который визуализирует навигационное меню и интегрирует его в приложение за счет обращения к этому компоненту из разделяемой компоновки. Подход подобного рода дает нам обычный класс C#, который может содержать любую необходимую прикладную логику и который можно подвергать модульному тестированию подобно любому другому классу. Такой способ удобен при создании небольших сегментов приложения с одновременным сохранением общего подхода MVC.

Создание навигационного компонента представления

Создадим папку по имени `Components`, которая по соглашению является местом хранения компонентов представлений, и добавим в нее файл класса под названием `NavigationMenuViewComponent.cs` с определением, показанным в листинге 9.6.

Листинг 9.6. Содержимое файла `NavigationMenuViewComponent.cs` из папки `Components`

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {

    public string Invoke() {
      return "Hello from the Nav View Component";
    }
  }
}
```

Метод `Invoke()` компонента представления вызывается, когда компонент применяется в представлении Razor, а результат, возвращаемый методом `Invoke()`, вставляется в HTML-разметку, отправляемую браузеру. Мы начали с простого компонента представления, который возвращает строку, но вскоре заменим его динамическим HTML-содержимым.

Список категорий должен присутствовать на всех страницах, поэтому мы собираемся использовать компонент представления в разделяемой компоновке, а не в отдельном представлении. Внутри компоновки компоненты представлений применяются через выражение `@await Component.InvokeAsync()`, как показано в листинге 9.7.

Листинг 9.7. Использование компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!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*" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <a class="navbar-brand" href="#">SPORTS STORE</a>
  </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>
```

Текст заполнителя заменен вызовом метода `Component.InvokeAsync()`. Аргументом метода является имя класса компонента без части `ViewComponent`, т.е. с помощью `NavigationMenu` указывается класс `NavigationMenuViewComponent`. Запустив приложение, можно заметить, что вывод из метода `InvokeAsync()` включен в HTML-разметку, отправляемую браузеру (рис. 9.2).

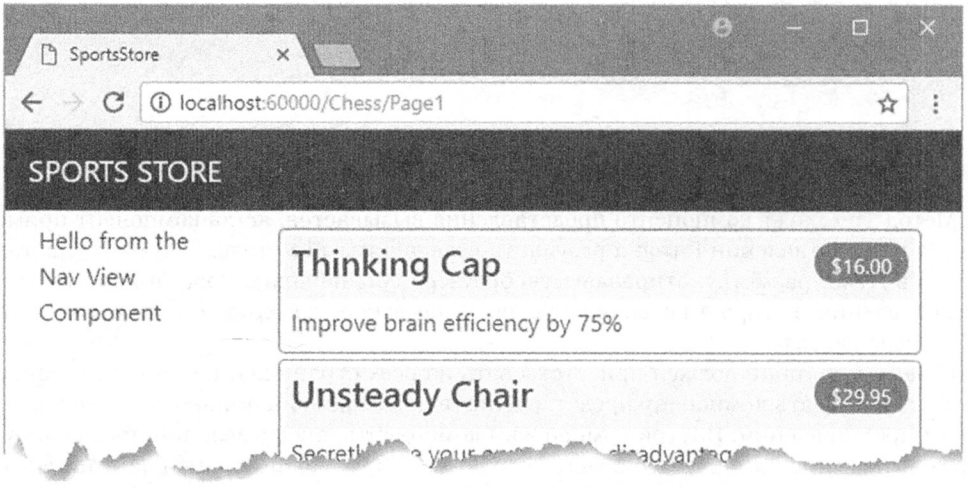

Рис. 9.2. Применение компонента представления

Генерация списков категорий

Теперь можно возвратиться к навигационному компоненту представления и сгенерировать реальный набор категорий. Построить HTML-разметку для категорий можно было бы программно, как делалось во вспомогательной функции дескриптора для страничных ссылок, но одно из преимуществ работы с компонентами представлений заключается в том, что они могут визуализировать частичные представления Razor. Это означает, что компонент представления можно использовать для генерации списка категорий и затем применить более выразительный синтаксис Razor для визуализации HTML-разметки, которая отобразит данный список. Первым делом нужно обновить компонент представления, как показано в листинге 9.8.

Листинг 9.8. Добавление категорий в файле `NavigationMenuViewComponent.cs` из папки `Components`

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

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {
    private IProductRepository repository;

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

    public IViewComponentResult Invoke() {
      return View(repository.Products
        .Select(x => x.Category)
        .Distinct()
        .OrderBy(x => x));
    }
  }
}
```

Конструктор, определенный в листинге 9.8, принимает аргумент типа `IProductRepository`. Когда инфраструктуре MVC необходимо создать экземпляр класса компонента представления, она отметит потребность в предоставлении этого аргумента и просмотрит конфигурацию в классе `Startup`, чтобы выяснить, какой объект реализации должен использоваться. Мы имеем дело с тем же самым средством внедрения зависимостей, которое применялось в контроллере из главы 8, и результат будет аналогичным — обеспечение компоненту представления доступа к данным без необходимости знать используемую реализацию хранилища, как описано в главе 18.

В методе `Invoke()` с помощью LINQ выбирается и упорядочивается набор категорий в хранилище, после чего он передается в качестве аргумента методу `View()`, который визуализирует стандартное частичное представление Razor. Детали частичного представления возвращаются из метода с применением объекта реализации `IViewComponentResult` (данный процесс будет подробно рассматриваться в главе 22).

Модульное тестирование: генерация списка категорий

Модульный тест, предназначенный для проверки возможности генерации списка категорий, относительно прост. Цель заключается в том, чтобы создать список, который отсортирован в алфавитном порядке и не содержит дубликатов. Проще всего это сделать, построив тестовые данные, которые *имеют* дублированные категории и *не* отсортированы должным образом, передав их вспомогательной функции дескриптора и установив утверждение, что данные были соответствующим образом очищены. Вот модульный тест, который определяется в новом файле класса по имени `NavigationMenuViewComponentTests.cs` внутри проекта `SportsStore.Tests`:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using SportsStore.Components;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

  public class NavigationMenuViewComponentTests {

    [Fact]
    public void Can_Select_Categories() {
      // Организация
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 2, Name = "P2", Category = "Apples"},
        new Product {ProductID = 3, Name = "P3", Category = "Plums"},
        new Product {ProductID = 4, Name = "P4", Category = "Oranges"},
      }).AsQueryable<Product>());

      NavigationMenuViewComponent target =
        new NavigationMenuViewComponent(mock.Object);

      // Действие - получение набора категорий
      string[] results = ((IEnumerable<string>)(target.Invoke()
        as ViewViewComponentResult).ViewData.Model).ToArray();

      // Утверждение
      Assert.True(Enumerable.SequenceEqual(new string[] { "Apples",
        "Oranges", "Plums" }, results));
    }
  }
}
```

Мы создаем имитированную реализацию хранилища, которая содержит повторяющиеся и несортированные категории. Затем мы устанавливаем утверждение о том, что дубликаты удалены и восстановлен алфавитный порядок.

Создание представления

При работе с представлениями, которые выбираются компонентами представлений, механизм Razor использует разнообразные соглашения. И стандартное имя представления, и местоположения, в которых ищется представление, отличаются от тех,

которые приняты для контроллеров. Создадим папку `Views/Shared/Components/NavigationMenu` и добавим в нее файл представления по имени `Default.cshtml` с содержимым, приведенным в листинге 9.9.

Листинг 9.9. Содержимое файла `Default.cshtml` из папки `Views/Shared/Components/NavigationMenu`

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
     Home
</a>

@foreach (string category in Model) {
  <a class="btn btn-block btn-secondary"
     asp-action="List"
     asp-controller="Product"
     asp-route-category="@category"
     asp-route-productPage="1">
       @category
  </a>
}
```

В представлении применяется один из встроенных классов вспомогательных функций дескрипторов (описанных в главах 24 и 25) для создания элементов `a`, атрибут `href` которых содержит URL, выбирающий определенную категорию товаров.

После запуска приложения появятся ссылки на категории (рис. 9.3). Щелчок на какой-то категории приводит к тому, что список товаров обновляется, отображая только товары выбранной категории.

Рис. 9.3. Генерация ссылок на категории с помощью компонента представления

Подсветка текущей категории

В настоящий момент пользователь не располагает какой-нибудь визуальной подсказкой о выбранной категории. Хотя, основываясь на товарах в списке, можно выдвинуть предположение относительно категории, гораздо лучше предоставить более наглядную визуальную обратную связь. Компоненты ASP.NET Core MVC, такие как контроллеры и компоненты представлений, могут получать информацию о текущем запросе, обращаясь к объекту контекста. Большую часть времени заботу о получении объекта контекста можно поручить базовым классам, которые используются для создания компонентов, подобно тому, как базовый класс `Controller` применяется для создания контроллеров.

Базовый класс `ViewComponent` не является исключением и обеспечивает доступ к объектам контекста через набор свойств. Одно из свойств называется `RouteData` и предоставляет информацию о том, как URL запроса был обработан системой маршрутизации.

В листинге 9.10 свойство `RouteData` используется для доступа к данным запроса, чтобы получить значение выбранной в текущий момент категории. Значение категории можно было бы передать представлению путем создания еще одного класса модели представления (и так бы делалось в реальном проекте), но ради разнообразия применим объект `ViewBag`, который был введен в главе 2.

Листинг 9.10. Передача выбранной категории в файле NavigationMenuViewComponent.cs из папки Components

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

namespace SportsStore.Components {

  public class NavigationMenuViewComponent : ViewComponent {
    private IProductRepository repository;

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

    public IViewComponentResult Invoke() {
      ViewBag.SelectedCategory = RouteData?.Values["category"];
      return View(repository.Products
        .Select(x => x.Category)
        .Distinct()
        .OrderBy(x => x));
    }
  }
}
```

Внутри метода `Invoke()` мы динамически создаем свойство `SelectedCategory` в объекте `ViewBag` и устанавливаем его значение равным значению текущей категории, которое получаем через объект контекста, возвращенный свойством `RouteData`. Как объяснялось в главе 2, `ViewBag` представляет собой динамический объект, который позволяет определять новые свойства, просто присваивая им значения.

Модульное тестирование: сообщение о выбранной категории

Для выполнения проверки того, что компонент представления корректно добавил детали о выбранной категории, в модульном тесте можно прочитать значение свойства `ViewBag`, которое доступно через класс `ViewViewComponentResult`, описанный в главе 22. Ниже показан модульный тест, добавленный в класс `NavigatioMenuViewComponentTests`:

```
...
[Fact]
public void Indicates_Selected_Category() {

  // Организация
  string categoryToSelect = "Apples";
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Apples"},
    new Product {ProductID = 4, Name = "P2", Category = "Oranges"},
  }).AsQueryable<Product>());

  NavigationMenuViewComponent target =
    new NavigationMenuViewComponent(mock.Object);
  target.ViewComponentContext = new ViewComponentContext {
    ViewContext = new ViewContext {
      RouteData = new RouteData()
    }
  };
  target.RouteData.Values["category"] = categoryToSelect;

  // Действие
  string result = (string)(target.Invoke() as
    ViewViewComponentResult).ViewData["SelectedCategory"];

  // Утверждение
  Assert.Equal(categoryToSelect, result);
}
...
```

Модульный тест снабжает компонент представления данными маршрутизации через свойство `ViewComponentContext`, посредством которого компоненты представлений получают все свои данные контекста. Свойство `ViewComponentContext` обеспечивает доступ к данным контекста, специфичным для представления, с помощью своего свойства `ViewContext`, которое в свою очередь обеспечивает доступ к информации о маршрутизации через собственное свойство `RouteData`. Большая часть кода в модульном тесте связана с созданием объектов контекста, которые будут предоставлять выбранную категорию таким же способом, как она бы предлагалась во время выполнения приложения, когда данные контекста предоставляются инфраструктурой ASP.NET Core MVC.

Теперь, когда доступна информация о том, какая категория выбрана, можно обновить представление, выбираемое компонентом представления, чтобы задействовать эту информацию, и изменить классы CSS, которые используются для стилизации ссылок, сделав представление текущей категории отличающимся от остальных категорий. В листинге 9.11 приведено изменение, внесенное в файл `Default.cshtml`.

Листинг 9.11. Подсветка текущей категории в файле `Default.cshtml` из папки `Views/Shared/Components/NavigationMenu`

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
     Home
</a>

@foreach (string category in Model) {
  <a class="btn btn-block
    @(category == ViewBag.SelectedCategory ? "btn-primary": "btn-secondary")"
    asp-action="List"
    asp-controller="Product"
    asp-route-category="@category"
    asp-route-productPage="1">
      @category
  </a>
}
```

С помощью выражения Razor внутри атрибута `class` мы применяем класс `btn-primary` к элементу, который представляет выбранную категорию, и класс `btn-secondary` к остальным элементам. Указанные классы применяют разные стили Bootstrap и делают активную кнопку визуально отличающейся (рис. 9.4).

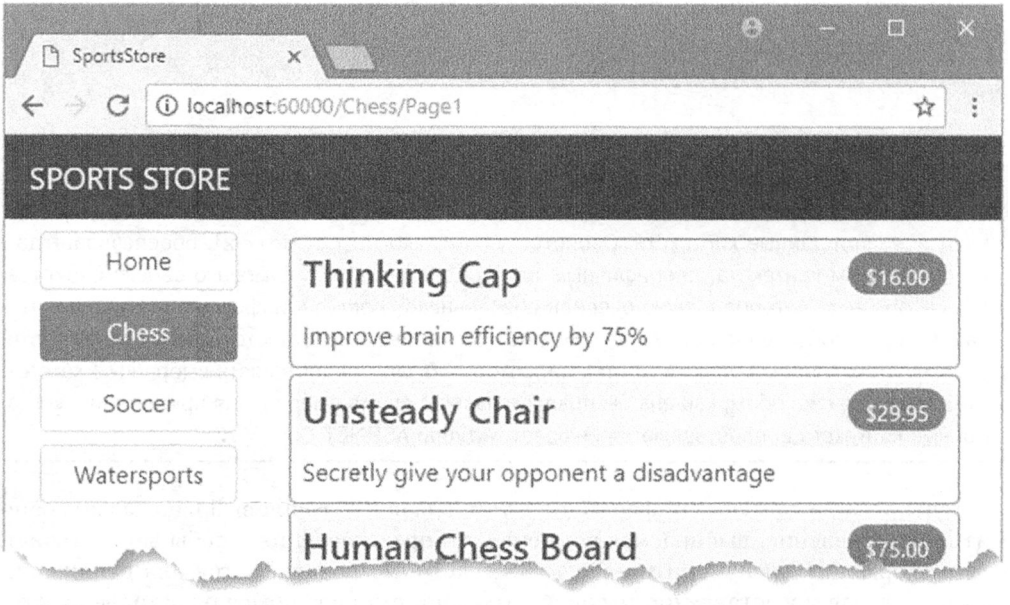

Рис. 9.4. Подсвечивание выбранной категории

Корректировка счетчика страниц

Мы должны скорректировать ссылки на страницы, чтобы они правильно работали, когда выбрана какая-то категория. В настоящий момент количество ссылок на страницы определяется общим числом товаров в хранилище, а не количеством товаров выбранной категории. Это значит, что пользователь может щелкнуть на ссылке для страницы 2 категории `Chess` и получить пустую страницу, поскольку товаров данной категории не хватает для заполнения второй страницы. Проблема демонстрируется на рис. 9.5.

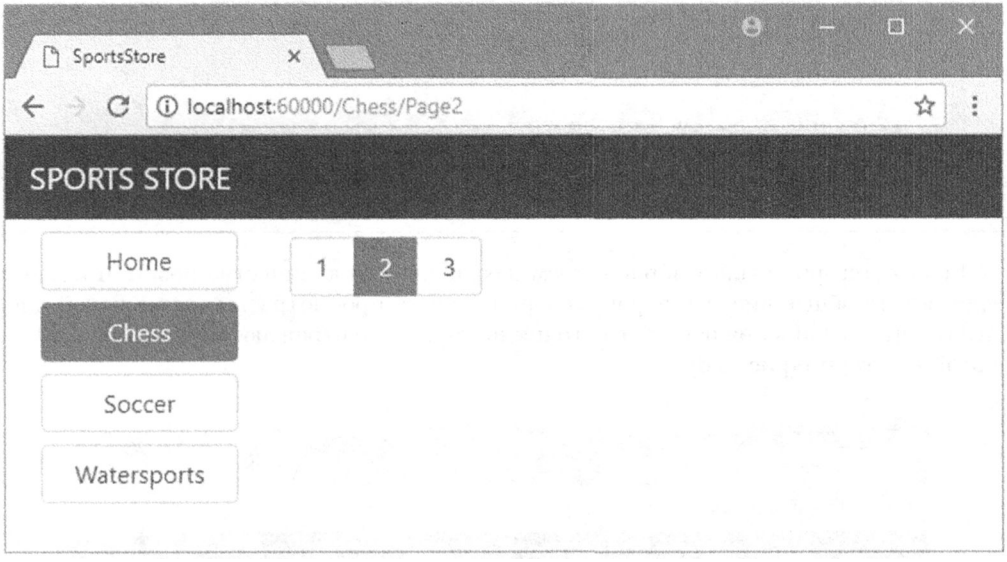

Рис. 9.5. Отображение некорректных ссылок на страницы, когда выбрана какая-то категория

Проблему можно устранить, модифицировав метод действия `List()` в контроллере `Product` так, чтобы при разбиении на страницы категории принимались во внимание (листинг 9.12).

Листинг 9.12. Создание данных о разбиении на страницы, учитывающих категории, в файле `ProductController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

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

    public ProductController(IProductRepository repo) {
      repository = repo;
    }
```

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

Если категория была выбрана, тогда возвращается количество позиций в ней, а если нет, то общее число товаров. Теперь во время просмотра товаров в какой-либо категории ссылки в нижней части страницы корректно отражают количество товаров в этой категории (рис. 9.6).

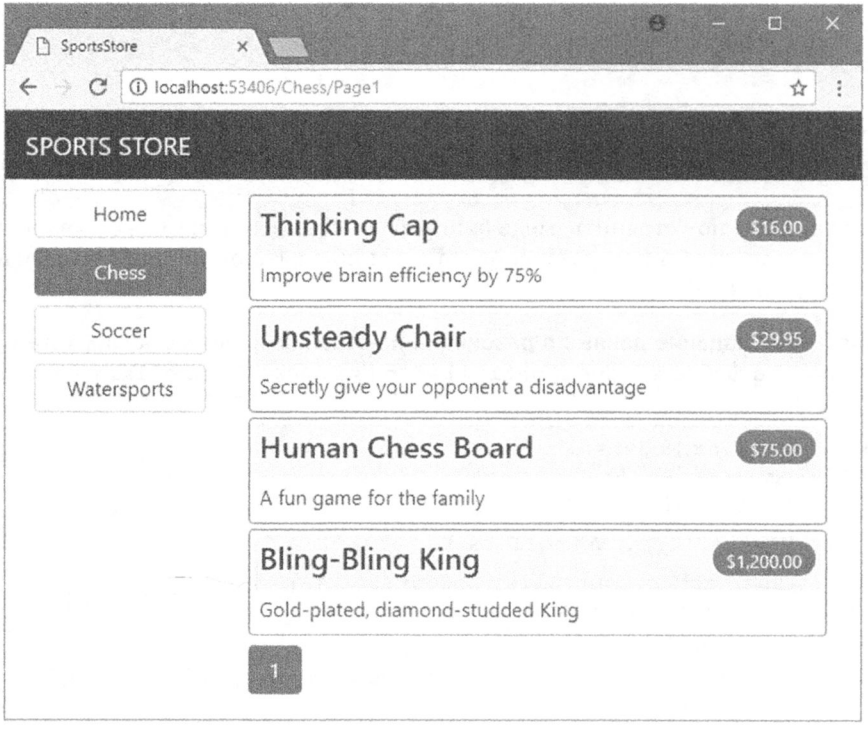

Рис. 9.6. Отображение ссылок на страницы с учетом выбранной категории

Модульное тестирование: счетчик товаров определенной категории

Протестировать возможность генерации корректных счетчиков товаров для различных категорий очень просто. Мы создадим имитированное хранилище, которое содержит известные данные в определенном диапазоне категорий, и затем вызовем метод действия `List()`, запрашивая каждую категорию по очереди. Вот модульный тест, добавленный в класс `ProductControllerTests`:

```
...
[Fact]
public void Generate_Category_Specific_Product_Count() {
  // Организация
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
    new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
    new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
    new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
    new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
  }).AsQueryable<Product>());

  ProductController target = new ProductController(mock.Object);
  target.PageSize = 3;

  Func<ViewResult, ProductsListViewModel> GetModel = result =>
    result?.ViewData?.Model as ProductsListViewModel;

  // Действие
  int? res1 = GetModel(target.List("Cat1"))?.PagingInfo.TotalItems;
  int? res2 = GetModel(target.List("Cat2"))?.PagingInfo.TotalItems;
  int? res3 = GetModel(target.List("Cat3"))?.PagingInfo.TotalItems;
  int? resAll = GetModel(target.List(null))?.PagingInfo.TotalItems;

  // Утверждение
  Assert.Equal(2, res1);
  Assert.Equal(2, res2);
  Assert.Equal(1, res3);
  Assert.Equal(5, resAll);
}
...
```

Обратите внимание, что в модульном тесте также вызывается метод `List()` без указания категории, чтобы удостовериться в правильности подсчета общего количества товаров.

Построение корзины для покупок

Приложение успешно расширяется, но продавать какие-либо товары до тех пор, пока не будет реализована корзина для покупок, не удастся. В настоящем разделе мы создадим корзину для покупок согласно иллюстрации на рис. 9.7. Если вы приобретали что-нибудь в электронных магазинах, то она должна выглядеть знакомой.

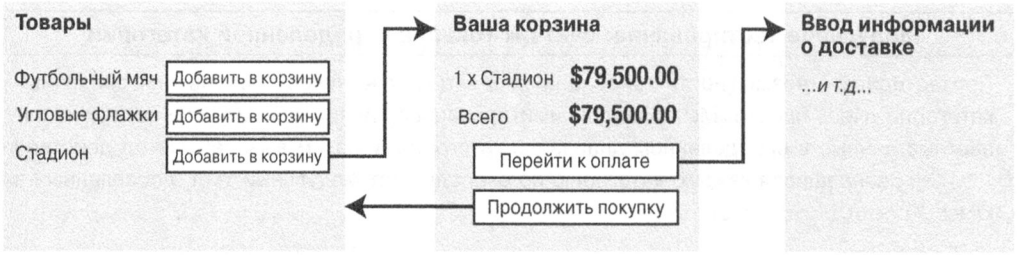

Рис. 9.7. Базовый поток корзины для покупок

Кнопка добавления в корзину (Add to cart) будет отображаться рядом с каждым товаром в каталоге. Щелчок на ней будет приводить к выводу сводки по товарам, которые уже были выбраны пользователем, включая их общую стоимость. В этой точке пользователь может с помощью кнопки продолжения покупки (Continue shopping) возвратиться в каталог товаров, а посредством кнопки перехода к оплате (Checkout now) сформировать заказ и завершить сеанс покупки.

Определение модели корзины

Начнем с добавления в папку `Models` файла класса по имени `Cart.cs` с определениями, показанными в листинге 9.13.

Листинг 9.13. Содержимое файла `Cart.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class Cart {
    private List<CartLine> lineCollection = new List<CartLine>();

    public virtual void AddItem(Product product, int quantity) {
      CartLine line = lineCollection
        .Where(p => p.Product.ProductID == product.ProductID)
        .FirstOrDefault();

      if (line == null) {
        lineCollection.Add(new CartLine {
          Product = product,
          Quantity = quantity
        });
      } else {
        line.Quantity += quantity;
      }
    }

    public virtual void RemoveLine(Product product) =>
      lineCollection.RemoveAll(l => l.Product.ProductID == product.ProductID);

    public virtual decimal ComputeTotalValue() =>
      lineCollection.Sum(e => e.Product.Price * e.Quantity);

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

    public virtual IEnumerable<CartLine> Lines => lineCollection;
  }
```

```
  public class CartLine {
    public int CartLineID { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
  }
}
```

Класс `Cart` использует класс `CartLine`, который определен в том же самом файле и представляет товар, выбранный пользователем, а также приобретаемое количество товара. Мы определили методы для добавления элемента в корзину, удаления элемента из корзины, вычисления общей стоимости элементов в корзине и очистки корзины путем удаления всех элементов. Мы также предоставили свойство, которое позволяет обратиться к содержимому корзины с применением `IEnumerable<CartLine>`. Все они легко реализуются с помощью кода C# и небольшой доли кода LINQ.

Модульное тестирование: проверка корзины

Класс `Cart` относительно прост, но в нем присутствует ряд важных линий поведения, которые должны корректно работать. Неправильно функционирующая корзина нарушит работу всего приложения SportsStore. Мы разобьем средства на части и протестируем их по отдельности. Для размещения тестов в проекте `SportsStore.Tests` создан новый файл по имени `CartTests.cs`.

Первая линия поведения относится к добавлению элемента в корзину. При первоначальном добавлении в корзину объекта `Product` должен быть добавлен новый экземпляр `CartLine`. Ниже представлен тестовый метод вместе с определением класса модульного тестирования.

```
using System.Linq;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

  public class CartTests {

    [Fact]
    public void Can_Add_New_Lines() {
      // Организация - создание нескольких тестовых товаров
      Product p1 = new Product { ProductID = 1, Name = "P1" };
      Product p2 = new Product { ProductID = 2, Name = "P2" };

      // Организация - создание новой корзины
      Cart target = new Cart();

      // Действие
      target.AddItem(p1, 1);
      target.AddItem(p2, 1);
      CartLine[] results = target.Lines.ToArray();

      // Утверждение
      Assert.Equal(2, results.Length);
      Assert.Equal(p1, results[0].Product);
      Assert.Equal(p2, results[1].Product);
    }
  }
}
```

Но если пользователь уже добавлял объект `Product` в корзину, тогда необходимо увеличить количество в соответствующем экземпляре `CartLine`, а не создавать новый. Вот модульный тест:

```
...
[Fact]
public void Can_Add_Quantity_For_Existing_Lines() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Действие
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);
  target.AddItem(p1, 10);
  CartLine[] results = target.Lines
    .OrderBy(c => c.Product.ProductID).ToArray();

  // Утверждение
  Assert.Equal(2, results.Length);
  Assert.Equal(11, results[0].Quantity);
  Assert.Equal(1, results[1].Quantity);
}
...
```

Также понадобится проверить, что пользователи имеют возможность менять свое решение и удалять товары из корзины. Эта линия поведения реализуется методом `RemoveLine()`. Модульный тест выглядит следующим образом:

```
...
[Fact]
public void Can_Remove_Line() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1" };
  Product p2 = new Product { ProductID = 2, Name = "P2" };
  Product p3 = new Product { ProductID = 3, Name = "P3" };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Организация - добавление некоторых товаров в корзину
  target.AddItem(p1, 1);
  target.AddItem(p2, 3);
  target.AddItem(p3, 5);
  target.AddItem(p2, 1);

  // Действие
  target.RemoveLine(p2);

  // Утверждение
  Assert.Equal(0, target.Lines.Where(c => c.Product == p2).Count());
  Assert.Equal(2, target.Lines.Count());
}
...
```

Далее проверяется линия поведения, связанная с возможностью вычисления общей стоимости элементов в корзине. Вот модульный тест:

```
...
[Fact]
public void Calculate_Cart_Total() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
  Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Действие
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);
  target.AddItem(p1, 3);
  decimal result = target.ComputeTotalValue();

  // Утверждение
  Assert.Equal(450M, result);
}
...
```

Последняя проверка очень проста. Мы должны удостовериться, что в результате очистки корзины ее содержимое корректно удаляется. Ниже показан модульный тест.

```
...
[Fact]
public void Can_Clear_Contents() {
  // Организация - создание нескольких тестовых товаров
  Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
  Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

  // Организация - создание новой корзины
  Cart target = new Cart();

  // Организация - добавление нескольких элементов в корзину
  target.AddItem(p1, 1);
  target.AddItem(p2, 1);

  // Действие - очистка корзины
  target.Clear();

  // Утверждение
  Assert.Equal(0, target.Lines.Count());
}
...
```

Временами, как в данном случае, код для тестирования функциональности класса получается намного длиннее и сложнее, чем код самого класса. Не допускайте, чтобы это приводило к отказу от написания модульных тестов. Дефекты в простых классах, особенно в тех, которые играют настолько важную роль, как `Cart` в приложении SportsStore, могут оказывать разрушительное воздействие.

Создание кнопок добавления в корзину

Нам необходимо модифицировать частичное представление `Views/Shared/ProductSummary.cshtml`, добавив кнопки к спискам товаров. В качестве подготовки добавим в папку `Infrastructure` файл класса по имени `UrlExtensions.cs` с определением расширяющего метода, приведенного в листинге 9.14.

Листинг 9.14. Содержимое файла `UrlExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

  public static class UrlExtensions {

    public static string PathAndQuery(this HttpRequest request) =>
      request.QueryString.HasValue
        ? $"{request.Path}{request.QueryString}"
        : request.Path.ToString();
  }
}
```

Расширяющий метод `PathAndQuery()` работает с классом `HttpRequest`, используемый в ASP.NET Core для описания HTTP-запроса. Расширяющий метод генерирует URL, по которому браузер будет возвращаться после обновления корзины, принимая во внимание строку запроса, если она есть. В листинге 9.15 в файл импортирования представлений добавляется пространство имен, которое содержит расширяющий метод, так что его можно применять в частичном представлении.

Листинг 9.15. Добавление пространства имен в файле `_ViewImports.cshtml` из папки `Views`

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

В листинге 9.16 показано частичное представление с описанием каждого товара, которое обновлено для включения кнопки Add To Cart (Добавить в корзину).

Листинг 9.16. Добавление кнопки в файле `ProductSummary.cshtml` из папки `Views/Shared`

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <form id="@Model.ProductID" asp-action="AddToCart"
      asp-controller="Cart" method="post">
    <input type="hidden" asp-for="ProductID" />
    <input type="hidden" name="returnUrl"
        value="@ViewContext.HttpContext.Request.PathAndQuery()" />
    <span class="card-text p-1">
      @Model.Description
      <button type="submit"
          class="btn btn-success btn-sm pull-right" style="float:right">
```

```
            Add To Cart
        </button>
    </span>
</form>
</div>
```

Мы добавили элемент `form`, содержащий скрытые элементы `input`, которые устанавливают значение `ProductID` из модели представления и URL, куда браузер должен возвращаться после обновления корзины. Элемент `form` и один из элементов `input` конфигурируются с использованием встроенных вспомогательных функций дескрипторов, что является удобным способом генерирования форм, которые содержат значения модели и нацелены на контроллеры и действия в приложении, как описано в главе 24. Во втором элементе `input` применяется расширяющий метод, созданный для установки URL возврата. Кроме того, добавлен элемент `button`, который будет отправлять форму приложению.

На заметку! Обратите внимание, что атрибут `method` элемента `form` установлен в `post`, что инструктирует браузер относительно отправки данных формы с использованием HTTP-метода `POST`. Вы можете это изменить и заставить форму применять HTTP-метод `GET`, но должны соблюдать осторожность. Спецификация HTTP требует, чтобы запросы `GET` были *идемпотентными*, т.е. они не должны приводить к изменениям, а добавление товара в корзину определенно считается изменением. Более подробно такая тема будет обсуждаться в главе 16, включая объяснение того, что может произойти, если проигнорировать требование идемпотентности запросов `GET`.

Включение поддержки сеансов

Мы собираемся сохранять детали корзины пользователя с использованием состояния сеанса, что представляет собой данные, которые хранятся на сервере и ассоциируются с последовательностью запросов, сделанных пользователем. Инфраструктура ASP.NET предлагает целый ряд разных способов хранения состояния сеанса, в том числе хранение его в памяти, что мы и будем применять. Преимуществом такого подхода является простота, но данные сеанса будут утеряны, когда приложение останавливается или перезапускается. Включение поддержки сеансов требует добавления в класс `Startup` служб и промежуточного программного обеспечения (листинг 9.17).

Листинг 9.17. Включение поддержки сеансов в файле `Startup.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
```

```
namespace SportsStore {
  public class Startup {
    public Startup(IConfiguration configuration) =>
      Configuration = configuration;

    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
      services.AddMemoryCache();
      services.AddSession();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseSession();
      app.UseMvc(routes => {

        // ...для краткости конфигурация маршрутизации не показана...

      });
      SeedData.EnsurePopulated(app);
    }
  }
}
```

Вызов метода `AddMemoryCache()` настраивает хранилище данных в памяти. Метод `AddSession()` регистрирует службы, используемые для доступа к данным сеанса, а метод `UseSession()` позволяет системе сеансов автоматически ассоциировать запросы с сеансами, когда они поступают от клиента.

Реализация контроллера для корзины

Для обработки щелчков на кнопках Add To Cart понадобится создать контроллер. Добавим в папку `Controllers` файл класса по имени `CartController.cs` с определением, представленным в листинге 9.18.

Листинг 9.18. Содержимое файла `CartController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
  public class CartController : Controller {
    private IProductRepository repository;

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

```
    public RedirectToActionResult AddToCart(int productId, string returnUrl) {
      Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
      if (product != null) {
        Cart cart = GetCart();
        cart.AddItem(product, 1);
        SaveCart(cart);
      }
      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 cart = GetCart();
        cart.RemoveLine(product);
        SaveCart(cart);
      }
      return RedirectToAction("Index", new { returnUrl });
    }
    private Cart GetCart() {
      Cart cart = HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();
      return cart;
    }
    private void SaveCart(Cart cart) {
      HttpContext.Session.SetJson("Cart", cart);
    }
  }
}
```

Относительно класса контроллера `CartController` необходимо сделать несколько замечаний. Для сохранения и извлечения объектов `Cart` применяется средство состояния сеанса ASP.NET, с которым и взаимодействует метод `GetCart()`. Промежуточное программное обеспечение, зарегистрированное в предыдущем разделе, использует cookie-наборы или переписывание URL, чтобы ассоциировать вместе множество запросов от определенного пользователя с целью формирования отдельного сеанса просмотра. Связанным средством является состояние сеанса, которое ассоциирует данные с сеансом. Ситуация идеально подходит для класса `Cart`: мы хотим, чтобы каждый пользователь имел собственную корзину, и она сохранялась между запросами. Данные, связанные с сеансом, удаляются по истечении времени существования сеанса (обычно из-за того, что пользователь не отправляет запрос какое-то время), т.е. управлять хранилищем или жизненным циклом объектов `Cart` не придется.

В методах действий `AddToCart()` и `RemoveFromCart()` применялись имена параметров, которые соответствуют именам элементов `input` в HTML-формах, созданных в представлении `ProductSummary.cshtml`. Это позволяет инфраструктуре MVC ассоциировать входящие переменные HTTP-запроса `POST` формы с параметрами и означает, что делать что-то самостоятельно для обработки формы не нужно. Такой процесс называется *привязкой модели* и с его помощью можно значительно упрощать классы контроллеров, как будет объясняться в главе 26.

Определение расширяющих методов состояния сеанса

Средство состояния сеанса в ASP.NET Core хранит только значения `int`, `string` и `byte[]`. Поскольку мы хотим сохранять объект `Cart`, необходимо определить расширяющие методы для интерфейса `ISession`, которые предоставят доступ к данным состояния сеанса с целью сериализации объектов `Cart` в формат JSON и их обратного преобразования. Добавим в папку `Infrastructure` файл класса по имени `SessionExtensions.cs` с определениями расширяющих методов, показанными в листинге 9.19.

Листинг 9.19. Содержимое файла `SessionExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace SportsStore.Infrastructure {

  public static class SessionExtensions {

    public static void SetJson(this ISession session, string key,
                               object value) {
      session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T GetJson<T>(this ISession session, string key) {
      var sessionData = session.GetString(key);
      return sessionData == null
        ? default(T) : JsonConvert.DeserializeObject<T>(sessionData);
    }
  }
}
```

При сериализации объектов в формат JSON (JavaScript Object Notation — система обозначений для объектов JavaScript) расширяющие методы полагаются на пакет Json.NET, который вы снова встретите в главе 20. Пакет Json.NET не требуется добавлять к проекту, потому что он уже используется "за кулисами" инфраструктурой MVC для поддержки средства заголовков JSON, которое описано в главе 21. (По адресу `https://www.newtonsoft.com/json` доступна информация о том, как работать с пакетом Json.NET напрямую.)

Расширяющие методы облегчают сохранение и извлечение объектов `Cart`. Для добавления объекта `Cart` к состоянию сеанса в контроллере применяется следующий вызов:

```
...
HttpContext.Session.SetJson("Cart", cart);
...
```

Свойство `HttpContext` определено в базовом классе `Controller`, от которого обычно унаследованы контроллеры, и возвращает объект `HttpContext`. Этот объект предоставляет данные контекста о запросе, который был получен, и ответе, находящемся в процессе подготовки. Свойство `HttpContext.Session` возвращает объект, реализующий интерфейс `ISession`. Данный интерфейс является именно тем типом, где мы определили метод `SetJson()`, принимающий аргументы, в которых указываются ключ и объект, подлежащий добавлению в состояние сеанса.

Расширяющий метод сериализирует объект и добавляет его в состояние сеанса, используя функциональность, которая лежит в основе интерфейса ISession.

Для извлечения объекта Cart применяется другой расширяющий метод, которому передается тот же самый ключ:ы

```
...
Cart cart = HttpContext.Session.GetJson<Cart>("Cart");
...
```

Параметр типа позволяет указать тип объекта, который ожидается извлечь; этот тип используется в процессе десериализации.

Отображение содержимого корзины

Финальное замечание о контроллере Cart касается того, что методы AddToCart() и RemoveFromCart() вызывают метод RedirectToAction(). Результатом будет отправка клиентскому браузеру HTTP-инструкции перенаправления, которая заставит браузер запросить новый URL. В данном случае браузер запросит URL, который вызывает метод действия Index() контроллера Cart.

Мы планируем реализовать метод Index() и применять его для отображения содержимого объекта Cart. Взглянув на рис. 9.7 еще раз, можно заметить, что это рабочий поток, инициируемый по щелчку пользователя на кнопке добавления в корзину.

Представлению, которое будет отображать содержимое корзины, необходимо передать две порции информации: объект Cart и URL для отображения в случае, если пользователь щелкнет на кнопке Continue shopping (Продолжить покупку). Для такой цели мы создадим простой класс модели представления. Добавим в папку Models/ViewModels проекта SportsStore файл класса по имени CartIndexViewModel.cs с содержимым, приведенным в листинге 9.20.

Листинг 9.20. Содержимое файла CartIndexViewModel.cs из папки Models/ViewModels

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

  public class CartIndexViewModel {
    public Cart Cart { get; set; }
    public string ReturnUrl { get; set; }
  }
}
```

Имея модель представления, можно реализовать метод действия Index() в классе CartController (листинг 9.21).

Листинг 9.21. Реализация метода действия Index() в файле CartController.cs из папки Controllers

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;
using SportsStore.Models.ViewModels;
```

```
namespace SportsStore.Controllers {
  public class CartController : Controller {
    private IProductRepository repository;

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

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

    // ...для краткости другие методы не показаны...
  }
}
```

Метод действия `Index()` извлекает объект `Cart` из состояния сеанса и использует его при создании объекта `CartIndexViewModel`, который затем передается методу `View()` для применения в качестве модели представления.

Последний шаг, относящийся к отображению содержимого корзины, предусматривает создание представления, которое будет визуализировать действие `Index`. Создадим папку `Views/Cart` и поместим в нее файл представления Razor по имени `Index.cshtml` с разметкой, приведенной в листинге 9.22.

Листинг 9.22. Содержимое файла `Index.cshtml` из папки `Views/Cart`

```
@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>
      </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>
```

Представление проходит по элементам в корзине и добавляет в HTML-таблицу строку для каждого элемента вместе со стоимостью и итоговой суммой по корзине. Классы, назначенные элементам, соответствуют стилям Bootstrap для таблиц и выравнивания текста.

В результате становится доступной базовая функциональность корзины для покупок. Во-первых, товары выводятся вместе с кнопками Add To Cart (Добавить в корзину), как показано на рис. 9.8.

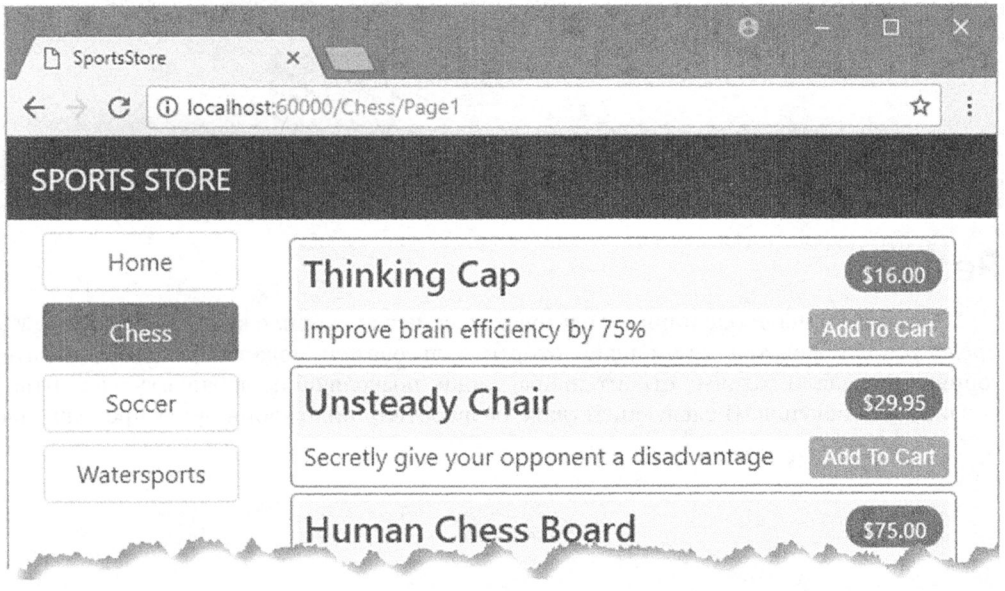

Рис. 9.8. Кнопки Add To Cart

Во-вторых, щелчок пользователя на кнопке Add To Cart приводит к добавлению соответствующего товара в его корзину и отображению сводной информации по корзине (рис. 9.9). Щелчок на кнопке Continue shopping (Продолжить покупку) возвращает пользователя на страницу товара, из которой произошел переход.

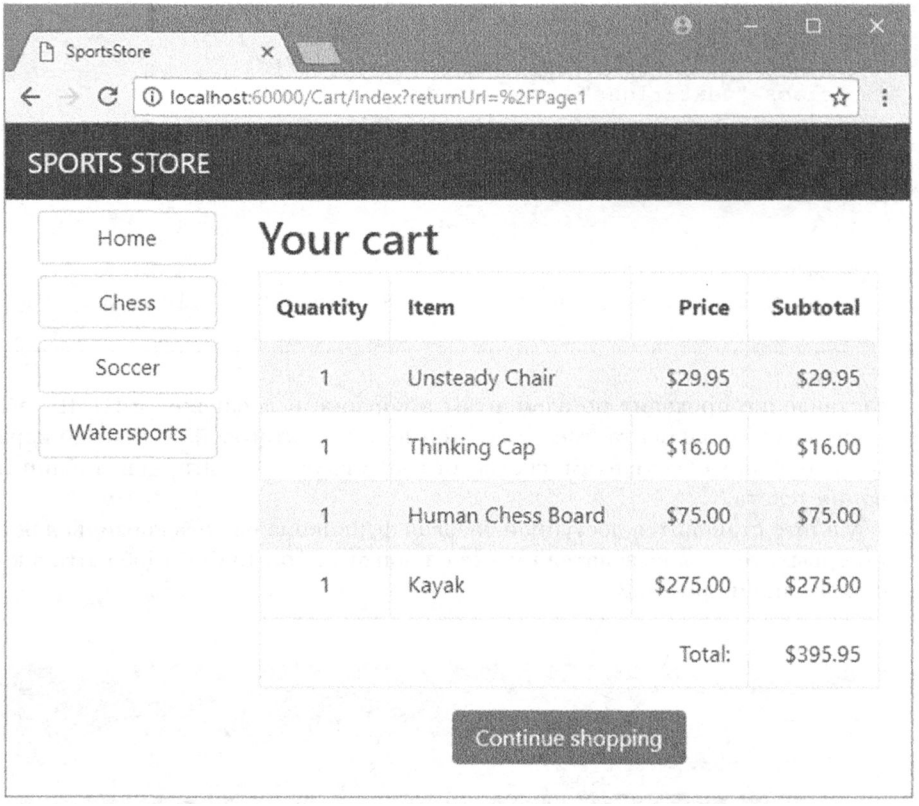

Рис. 9.9. Отображение содержимого корзины для покупок

Резюме

В главе мы начали расширять пользовательские части приложения SportsStore. Мы предоставили средства, с помощью которых пользователь может переходить по категориям, и создали базовые строительные блоки, позволяющие добавлять элементы в корзину для покупок. В следующей главе разработка приложения будет продолжена.

ГЛАВА 10

SportsStore: завершение построения корзины для покупок

В настоящей главе мы продолжим строить пример приложения SportsStore. В предыдущей главе мы добавили базовую поддержку корзины для покупок, а теперь собираемся усовершенствовать и завершить создание этой функциональности.

Усовершенствование модели корзины с помощью службы

В предыдущей главе был определен класс модели `Cart` и продемонстрирована возможность его сохранения с использованием средства состояния сеанса, что давало возможность пользователю формировать набор товаров для покупки. Обязанность по управлению постоянством класса `Cart` возлагалась на контроллер `Cart`, в котором явно определялись методы для получения и сохранения объектов `Cart`.

Проблема такого подхода в том, что нам пришлось дублировать код для получения и сохранения объектов `Cart` во всех компонентах, которые его применяли. В текущем разделе мы воспользуемся средством служб, которое находится в самой основе инфраструктуры ASP.NET Core, чтобы упростить способ управления объектами `Cart`, освобождая индивидуальные компоненты, такие как контроллер `Cart`, от необходимости напрямую иметь дело с деталями.

Службы чаще всего применяются для сокрытия деталей реализации интерфейсов от компонентов, которые от них зависят. Вы видели пример, когда создавалась служба для интерфейса `IProductRepository`, которая позволила гладко заменить фиктивный класс хранилища реальным хранилищем Entity Framework Core. Но службы могут использоваться для решения множества других задач, а также применяться для придания и изменения формы приложения, даже если работа производится с конкретными классами наподобие `Cart`.

Создание класса корзины, осведомленного о хранилище

Первым шагом по приведению в порядок способа использования класса `Cart` будет создание подкласса, который осведомлен о том, как сохранять самого себя с применением состояния сеанса. Добавим в папку `Models` файл класса по имени `SessionCart.cs` и поместим в него определение, показанное в листинге 10.1.

Листинг 10.1. Содержимое файла `SessionCart.cs` из папки `Models`

```
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");
    }
  }
}
```

Класс `SessionCart` является производным от класса `Cart` и переопределяет методы `AddItem()`, `RemoveLine()` и `Clear()`, так что они вызывают базовые реализации и затем сохраняют обновленное состояние в сеансе, используя расширяющие методы интерфейса `ISession`, которые были определены в главе 9. Статический метод `GetCart()` — это фабрика для создания объектов `SessionCart` и снабжения их объектом реализации `ISession`, чтобы они могли себя сохранять.

Получение объекта реализации `ISession` несколько затруднено. Мы должны получить экземпляр службы `IHttpContextAccessor`, который предоставит доступ к объекту `HttpContext`, а тот в свою очередь — к объекту реализации `ISession`. Такой непрямой подход требуется из-за того, что сеанс не предоставляется как обычная служба.

Регистрация службы

Следующий шаг заключается в создании службы для класса `Cart`. Цель в том, чтобы удовлетворять запросы для объектов `Cart` выдачей объектов `SessionCart`, которые будут сохранять себя самостоятельно. Создание службы иллюстрируется в листинге 10.2.

Листинг 10.2. Создание службы корзины в файле `Startup.cs` из папки `SportsStore`

```
...
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();
}
...
```

Метод `AddScoped()` указывает, что для удовлетворения связанных запросов к экземплярам `Cart` должен применяться один и тот же объект. Способ связывания запросов может быть сконфигурирован, но по умолчанию это значит, что в ответ на любой запрос экземпляра `Cart` со стороны компонентов, которые обрабатывают тот же самый HTTP-запрос, будет выдаваться один и тот же объект.

Вместо предоставления методу `AddScoped()` отображения между типами, как делалось для хранилища, указывается лямбда-выражение, которое будет выполняться для удовлетворения запросов к `Cart`. Лямбда-выражение получает коллекцию служб, которые были зарегистрированы, и передает ее методу `GetCart()` класса `SessionCart`. В результате запросы для службы `Cart` будут обрабатываться путем создания объектов `SessionCart`, которые сериализируют сами себя как данные сеанса, когда они модифицируются.

Мы также добавили службу с использованием метода `AddSingleton()`, который указывает, что всегда должен применяться один и тот же объект. Созданная служба сообщает инфраструктуре MVC о том, что когда требуются реализации интерфейса `IHttpContextAccessor`, необходимо использовать класс `HttpContextAccessor`. Данная служба обязательна, поэтому в классе `SessionCart` можно получать доступ к текущему сеансу, как делалось в листинге 10.1.

Упрощение контроллера `Cart`

Преимущество создания службы такого вида связано с тем, что она позволит упростить контроллеры, в которых применяются объекты `Cart`. В листинге 10.3 приведен переделанный класс `CartController`, где задействована новая служба.

Листинг 10.3. Использование службы `Cart` в файле `CartController.cs` из папки `Controllers`

```
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 ViewResult 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 });
    }
  }
}
```

Класс `CartController` указывает на то, что он нуждается в объекте `Cart`, за счет объявления аргумента конструктора. Это позволяет удалить методы, которые читают и записывают данные в сеанс, а также код, требующийся для записи обновлений. Результатом оказывается контроллер, который не только проще, но и более сосредоточен на своей роли в приложении, не беспокоясь о том, как объекты `Cart` создаются либо хранятся. И поскольку службы доступны по всему приложению, любой компонент может получать корзину пользователя с применением одного и того же приема.

Завершение функциональности корзины

После ввода службы `Cart` наступило время завершить построение функциональности корзины, добавив два новых средства. Первое средство позволит пользователю удалять элемент из корзины. Второе средство даст возможность отображать итоговую информацию по корзине в верхней части страницы.

Удаление элементов из корзины

Мы уже определили и протестировали метод действия `RemoveFromCart()` в контроллере, а потому для предоставления пользователям возможности удаления элементов достаточно лишь открыть доступ к данному методу в представлении, доба-

вив кнопки Remove (Удалить) ко всем строкам в итоговой информации по корзине. Изменения, которые понадобится внести в файл `Views/Cart/Index.cshtml`, показаны в листинге 10.4.

Листинг 10.4. Добавление кнопок Remove в файле `Index.cshtml` из папки `Views/Cart`

```
@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>
```

Мы добавили к каждой строке таблицы новый столбец, содержащий элемент `form` со скрытыми элементами `input`, которые указывают товар, подлежащий удалению, и URL возврата, а также кнопку для отправки формы.

Чтобы увидеть кнопки Remove в работе, понадобится запустить приложение и добавить несколько элементов в корзину для покупок. Поскольку корзина уже обладает функциональностью удаления элементов, ее можно протестировать, щелкнув на одной из новых кнопок (рис. 10.1).

Рис. 10.1. Удаление элемента из корзины для покупок

Добавление виджета с итоговой информацией по корзине

Мы имеем функционирующую корзину, но еще должны определить способ ее встраивания в пользовательский интерфейс. Пользователи могут выяснять, что находится в их корзинах, только за счет просмотра экрана со сводкой по корзине. Но попасть на этот экран можно только путем добавления нового элемента в корзину.

Для решения упомянутой задачи мы создадим виджет (графический элемент) с итоговой информацией по содержимому корзины, щелчок на котором будет приводить к отображению товаров, находящихся в корзине, и сделаем его доступным в любом месте приложения. Он будет реализован почти так же, как виджет для навигации — в виде компонента представления, вывод которого может включаться в разделяемую компоновку Razor.

Добавление пакета Font Awesome

В качестве части итоговой информации по корзине мы будем отображать кнопку, которая позволит пользователю перейти к оплате. Вместо отображения каких-либо слов на кнопке мы хотим использовать символ корзины. При отсутствии художественных навыков можно применить пакет с открытым кодом Font Awesome, предлагающий великолепный набор значков, которые допускается интегрировать в приложения как шрифты, где каждый символ шрифта представляет собой отдельное изображение. Получить дополнительные сведения о пакете Font Awesome, а также просмотреть содержащиеся в нем значки, можно по адресу `https://fontawesome.com/`.

Выберем проект SportsStore и щелкнем на кнопке Show All Items (Показать все элементы) в верхней части окна Solution Explorer, чтобы отобразить файл bower.json. Добавим пакет Font Awesome в раздел dependencies (листинг 10.5).

Листинг 10.5. Добавление пакета Font Awesome в файле bower.json из папки SportsStore

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

После сохранения файла bower.json среда Visual Studio с помощью инструмента Bower загрузит и установит пакет Font Awesome в папку wwwroot/lib/fontawesome.

Создание класса компонента представления и представления

Добавим в папку Components файл класса по имени CartSummaryViewComponent.cs и определим в нем компонент представления, показанный в листинге 10.6.

Листинг 10.6. Содержимое файла CartSummaryViewComponent.cs из папки 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);
    }
  }
}
```

Компонент представления CartSummaryViewComponent способен задействовать в своих интересах службу, созданную ранее в главе для получения объекта Cart, принимая ее как аргумент конструктора. Результатом оказывается простой компонент представления, который передает объект Cart методу View(), чтобы сгенерировать фрагмент HTML-разметки для включения в компоновку. Чтобы создать компоновку, создадим папку Views/Shared/Components/CartSummary, добавим в нее файл представления Razor по имени Default.cshtml и поместим в него разметку, приведенную в листинге 10.7.

Листинг 10.7. Содержимое файла `Default.cshtml` из папки `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-controller="Cart" asp-action="Index"
     sp-route-returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">
       <i class="fa fa-shopping-cart"></i>
  </a>
</div>
```

Представление отображает кнопку со значком корзины Font Awesome и когда в корзине присутствуют товары, предоставляет снимок, который сообщает количество элементов и общую сумму. Теперь, имея компонент представления и представление, можно модифицировать разделяемую компоновку, чтобы итоговая информация по корзине включалась в ответы, генерируемые контроллерами приложения (листинг 10.8).

Листинг 10.8. Добавление итоговой информации по корзине в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!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>
```

Запустив приложение, можно увидеть итоговую информацию по корзине. Когда корзина пуста, отображается только кнопка Continue shopping (Продолжить покупки). По мере добавления элементов в корзину выводится их количество и общая стоимость (рис. 10.2). Благодаря такому дополнению пользователь знает, какие товары находятся в корзине, и получает очевидный путь для перехода к оплате покупок.

Рис. 10.2. Отображение итоговой информации по корзине

Отправка заказов

Итак, мы добрались до финального пользовательского средства приложения SportsStore: возможности перехода к оплате и оформлению заказа. В последующих разделах мы расширим модель предметной области, чтобы обеспечить поддержку получения от пользователя подробной информации о доставке, и добавим в приложение обработку этой информации.

Создание класса модели

Добавим в папку `Models` файл класса по имени `Order.cs` и приведем его содержимое в соответствие с листингом 10.9. Класс `Order` будет использоваться для представления информации о доставке пользователю.

Листинг 10.9. Содержимое файла `Order.cs` из папки `Models`

```
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")]
                            // Введите имя
    public string Name { get; set; }
    [Required(ErrorMessage = "Please enter the first address line")]
                            // Введите первую строку адреса
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }
    [Required(ErrorMessage = "Please enter a city name")]
                            // Введите название города
    public string City { get; set; }
    [Required(ErrorMessage = "Please enter a state name")]
                            // Введите название штата
    public string State { get; set; }
    public string Zip { get; set; }
    [Required(ErrorMessage = "Please enter a country name")]
                            // Введите название страны
    public string Country { get; set; }
    public bool GiftWrap { get; set; }
  }
}
```

Здесь применяются атрибуты проверки достоверности из пространства имен `System.ComponentModel.DataAnnotations`, как мы делали в главе 2. Проверка достоверности подробно рассматривается в главе 27.

Кроме того, в классе `Order` используется атрибут `BindNever`, который предотвращает предоставление пользователем значений для снабженных этим атрибутом свойств в HTTP-запросе. Такая возможность системы привязки моделей, описанная в главе 26, останавливает применение инфраструктурой MVC значений из HTTP-запроса для заполнения конфиденциальных или важных свойств модели.

Добавление реализации процесса оплаты

Наша цель заключается в том, чтобы обеспечить пользователям возможность ввода информации о доставке и отправки заказа. Для начала потребуется добавить к представлению итоговой информации по корзине кнопку Checkout (Перейти к оплате). Изменения, которые необходимо внести в файл `Views/Cart/Index.cshtml`, показаны в листинге 10.10.

Листинг 10.10. Добавление кнопки Checkout в файле `Index.cshtml` из папки `Views/Cart`

```
...
<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>
...
```

Изменения обеспечивают генерацию ссылки, стилизованной в виде кнопки, щелчок на которой приводит к вызову метода действия `Checkout()` контроллера `Order`, создаваемого в следующем разделе. На рис. 10.3 показано, как выглядит кнопка Checkout.

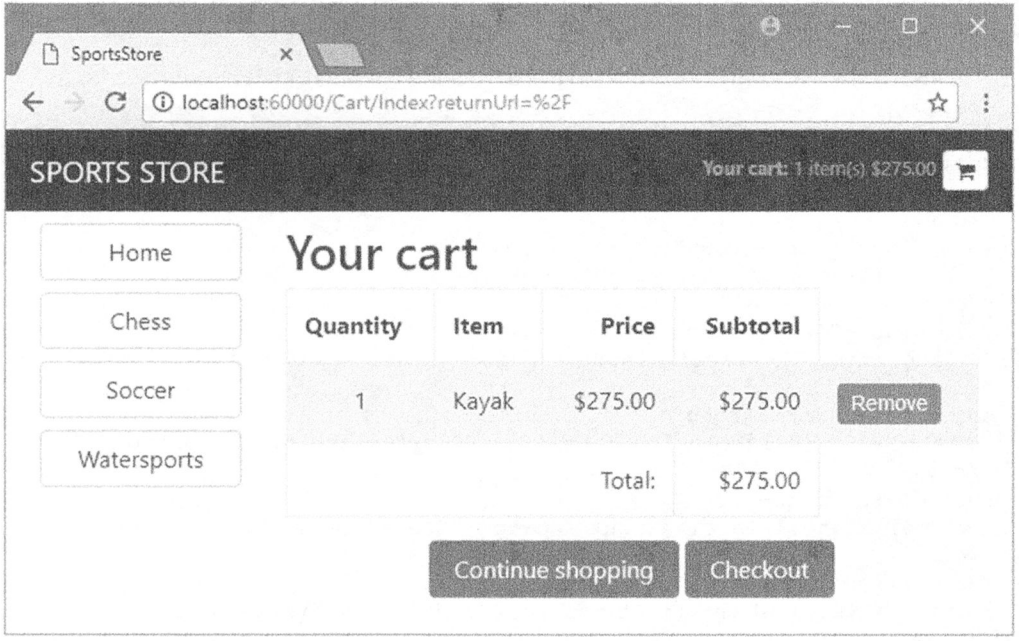

Рис. 10.3. Кнопка Checkout

Теперь понадобится определить контроллер `Order`. Добавим в папку `Controllers` файл класса по имени `OrderController.cs` с определением, приведенным в листинге 10.11.

Листинг 10.11. Содержимое файла `OrderController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class OrderController : Controller {

    public ViewResult Checkout() => View(new Order());
  }
}
```

Метод `Checkout()` возвращает стандартное представление и передает новый объект `ShippingDetails` в качестве модели представления. Чтобы создать представление, создадим папку `Views/Order` и поместим в нее файл представления Razor по имени `Checkout.cshtml` с разметкой, показанной в листинге 10.12.

Листинг 10.12. Содержимое файла `Checkout.cshtml` из папки `Views/Order`

```
@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>
```

Для каждого свойства в модели мы создали элементы `label` и `input` для пользовательского ввода, сформатированные с помощью Bootstrap. Атрибут `asp-for` в элементах `input` обрабатывается встроенной вспомогательной функцией дескриптора, которая генерирует атрибуты `type`, `id`, `name` и `value` на основе указанного свойства модели, как объясняется в главе 24.

Чтобы увидеть результат добавления нового метода действия и представления (рис. 10.4), запустим приложение, щелкнем на кнопке со значком корзины в верхней части страницы и затем на кнопке Checkout (Оплата). Попасть на данное представление можно также, запросив URL вида `/Order/Checkout`.

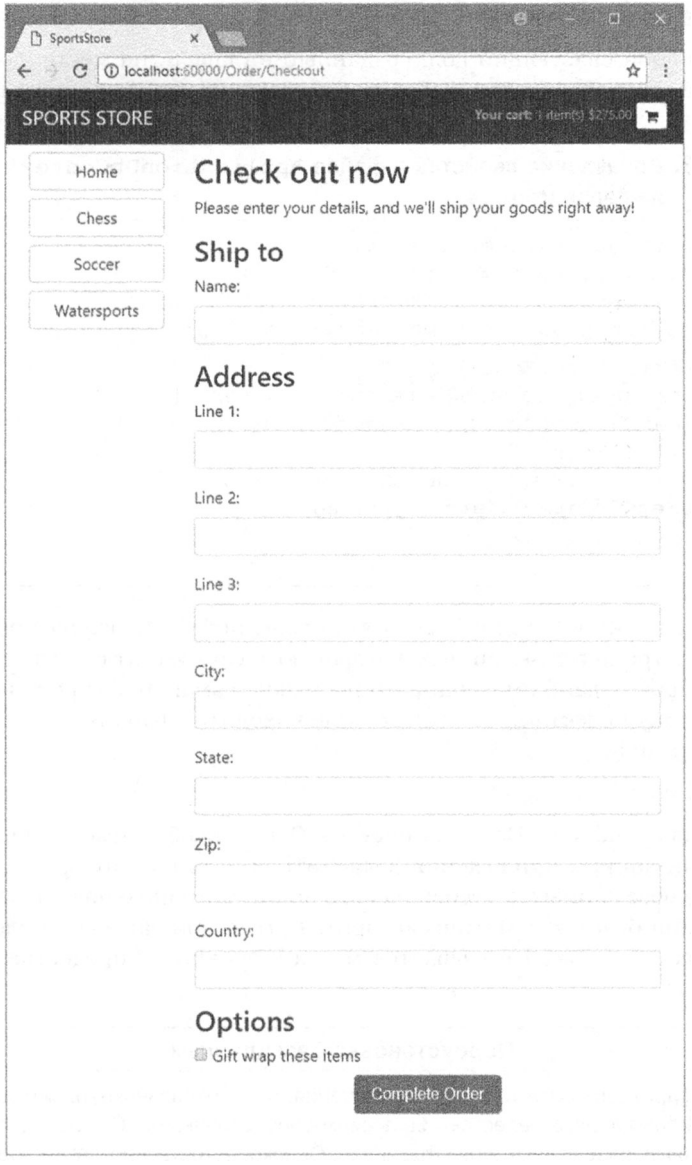

Рис. 10.4. Форма для сбора деталей о доставке

Реализация обработки заказов

Мы будем обрабатывать заказы, записывая их в базу данных. Разумеется, большинство сайтов электронной коммерции на этом не останавливаются, но мы не станем заниматься обработкой кредитных карт или других форм оплаты. Чтобы сосредоточиться на инфраструктуре MVC, вполне достаточно простого сохранения в базе данных.

Расширение базы данных

Когда основной связующий код, созданный в главе 8, на месте, добавить в базу данных новый вид модели легко. Добавим в класс контекста базы данных новое свойство, как показано в листинге 10.13.

Листинг 10.13. Добавление свойства в файле `ApplicationDbContext.cs` из папки `Models`

```
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; }
  }
}
```

Такого изменения достаточно для того, чтобы инфраструктура Entity Framework Core создала миграцию базы данных, которая позволит объектам `Order` сохраняться в базе данных. Для создания миграции откроем окно командной строки или PowerShell, перейдем в папку проекта `SportsStore` (где находится файл `Startup.cs`) и введем следующую команду:

```
dotnet ef migrations add Orders
```

Эта команда сообщает Entity Framework Core о необходимости получить новый снимок модели данных приложения, выяснить его отличия от предыдущей версии, хранящейся в базе данных, и сгенерировать новую миграцию под названием `Orders`. Новая миграция будет автоматически применяться при запуске приложения, потому что в классе `SeedData` вызывается метод `Migrate()`, предоставляемый Entity Framework Core.

Переустановка базы данных

Если в модель приходится часто вносить изменения, то может возникнуть ситуация, когда миграции и схема базы данных перестают быть синхронизированными. Самое простое, что можно сделать — удалить базу данных и начать заново. Однако, естественно, такой подход приемлем только на стадии разработки, потому что все сохраненные данные будут утрачены.

Чтобы удалить базу данных, необходимо ввести приведенную ниже команду, находясь в папке проекта `SportsStore`:

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

После того, как база данных удалена, ввод следующей команды в папке проекта `SportsStore` приводит к построению базы данных заново и применению созданных ранее миграций:

```
dotnet ef database update
```

В результате база данных переустанавливается, точно отражая модель, что дает возможность возвратиться к разработке приложения.

Создание хранилища заказов

Чтобы предоставить доступ к объектам `Order`, мы последуем тому же самому шаблону, который использовался для хранилища товаров. Добавим в папку `Models` новый файл по имени `IOrderRepository.cs` и определим в нем интерфейс, показанный в листинге 10.14.

Листинг 10.14. Содержимое файла `IOrderRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {

  public interface IOrderRepository {

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

Для реализации интерфейса хранилища заказов добавим в папку `Models` файл класса по имени `EFOrderRepository.cs` с определением, представленным в листинге 10.15.

Листинг 10.15. Содержимое файла `EFOrderRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class EFOrderRepository : IOrderRepository {
    private ApplicationDbContext context;

    public EFOrderRepository(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();
    }
  }
}
```

Класс `EFOrderRepository` реализует интерфейс `IOrderRepository` с применением Entity Framework Core, позволяя извлекать набор сохраненных объектов `Order` и создавать либо изменять заказы.

Особенности хранилища заказов

Реализация хранилища для заказов в листинге 10.15 требует небольшой дополнительной работы. Инфраструктуру Entity Framework Core необходимо проинструктировать о загрузке связанных данных, если они охватывают несколько таблиц. В листинге 10.15 с помощью методов `Include()` и `ThenInclude()` указано, что когда объект `Order` читается из базы данных, то также должна загружаться коллекция, ассоциированная со свойством `Lines`, наряду с объектами `Product`, которые связаны с элементами коллекции:

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

Такой подход гарантирует получение всех нужных объектов данных, не выполняя запросы и не собирая данные напрямую.

Дополнительный шаг требуется и при сохранении объекта `Order` в базе данных. Когда данные корзины пользователя десериализируются из состояния сеанса, пакет JSON создает новые объекты, не известные инфраструктуре Entity Framework Core, которая затем пытается записать все объекты в базу данных. В случае объектов `Product` это означает, что инфраструктура Entity Framework Core попытается записать объекты, которые уже были сохранены, что приведет к ошибке. Во избежание проблемы мы уведомляем Entity Framework Core о том, что объекты существуют и не должны сохраняться в базе данных до тех пор, пока они не будут модифицированы:

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

В результате инфраструктура Entity Framework Core не будет пытаться записывать десериализированные объекты `Product`, которые ассоциированы с объектом `Order`.

В листинге 10.16 хранилище заказов регистрируется как служба внутри метода `ConfigureServices()` класса `Startup`.

Листинг 10.16. Регистрация службы хранилища заказов в файле `Startup.cs` из папки `SportsStore`

```
...
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();
}
...
```

Завершение построения контроллера Order

Для завершения класса OrderController понадобится модифицировать конструктор так, чтобы он получал службы, требующиеся ему для обработки заказа, и добавить новый метод действия, который будет обрабатывать HTTP-запрос POST формы, когда пользователь щелкает на кнопке Complete order (Завершить заказ). Оба изменения показаны в листинге 10.17.

Листинг 10.17. Завершение контроллера в файле OrderController.cs из папки Controllers

```
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 ViewResult Checkout() => View(new Order());
    [HttpPost]
    public IActionResult Checkout(Order order) {
      if (cart.Lines.Count() == 0) {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
                                    // Корзина пуста!
      }
      if (ModelState.IsValid) {
        order.Lines = cart.Lines.ToArray();
        repository.SaveOrder(order);
        return RedirectToAction(nameof(Completed));
      } else {
        return View(order);
      }
    }
    public ViewResult Completed() {
      cart.Clear();
      return View();
    }
  }
}
```

Метод действия Checkout() декорирован атрибутом HttpPost, т.е. он будет вызываться для запроса POST — в этом случае, когда пользователь отправляет форму. Мы снова полагаемся на систему привязки моделей, так что можно получить объект Order, дополнить его данными из объекта Cart и сохранить в хранилище.

Инфраструктура MVC контролирует ограничения проверки достоверности, которые были применены к классу Order посредством атрибутов аннотаций данных, и через свойство ModelState сообщает методу действия о любых проблемах. Чтобы выяснить, есть ли проблемы, мы проверяем свойство ModelState.IsValid. Мы вызы-

ваем метод `ModelState.AddModelError()` для регистрации сообщения об ошибке, если в корзине нет элементов. Вскоре мы объясним, как отображать такие сообщения об ошибках, а более подробное описание привязки моделей и проверки достоверности будет представлено в главах 27 и 28.

Модульное тестирование: обработка заказа

Чтобы выполнить модульное тестирование класса `OrderController`, необходимо проверить поведение версии `POST` метода `Checkout()`. Хотя метод выглядит коротким и простым, использование привязки моделей MVC означает наличие многих вещей, происходящих "за кулисами", которые должны быть протестированы.

Мы хотим обрабатывать заказ, только если в корзине присутствуют элементы *и* пользователь предоставил достоверные детали о доставке. При любых других обстоятельствах пользователю должно быть сообщено об ошибке. Вот первый тестовый метод, который определен в файле класса по имени `OrderControllerTests.cs` внутри проекта `SportsStore.Tests`:

```
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() {
      // Организация - создание имитированного хранилища заказов
      Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
      // Организация - создание пустой корзины
      Cart cart = new Cart();
      // Организация - создание заказа
      Order order = new Order();
      // Организация - создание экземпляра контроллера
      OrderController target = new OrderController(mock.Object, cart);
      // Действие
      ViewResult result = target.Checkout(order) as ViewResult;
      // Утверждение - проверка, что заказ не был сохранен
      mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
      // Утверждение - проверка, что метод возвращает стандартное представление
      Assert.True(string.IsNullOrEmpty(result.ViewName));
      // Утверждение - проверка, что представлению передана
      // недопустимая модель
      Assert.False(result.ViewData.ModelState.IsValid);
    }
  }
}
```

Тест проверяет отсутствие возможности перехода к оплате при пустой корзине. Мы удостоверяемся, что метод `SaveOrder()` имитированной реализации `IOrderRepository` никогда не вызывается, что метод возвращает стандартное представление (которое повторно отобразит введенные пользователем данные, давая ему шанс откорректировать их) и что состояние модели, передаваемое представлению, помечено как недопустимое. Это может выглядеть как излишне ограничивающий набор утверждений, но для проверки правильности

поведения нужны все три утверждения. Следующий тестовый метод работает в основном так же, но внедряет в модель представления ошибку, эмулирующую проблему, о которой сообщает средство привязки модели (что должно происходить в производственной среде, когда пользователь вводит некорректные данные о доставке):

```
...
[Fact]
public void Cannot_Checkout_Invalid_ShippingDetails() {
  // Организация - создание имитированного хранилища заказов
  Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
  // Организация - создание корзины с одним элементом
  Cart cart = new Cart();
  cart.AddItem(new Product(), 1);
  // Организация - создание экземпляра контроллера
  OrderController target = new OrderController(mock.Object, cart);
  // Организация - добавление ошибки в модель
  target.ModelState.AddModelError("error", "error");
  // Действие - попытка перехода к оплате
  ViewResult result = target.Checkout(new Order()) as ViewResult;
  // Утверждение - проверка, что заказ не был сохранен
  mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
  // Утверждение - проверка, что метод возвращает стандартное представление
  Assert.True(string.IsNullOrEmpty(result.ViewName));
  // Утверждение - проверка, что представлению передается недопустимая модель
  Assert.False(result.ViewData.ModelState.IsValid);
}
...
```

Удостоверившись в том, что пустая корзина или некорректные данные о доставке предотвращают сохранение заказа, необходимо проверить, что нормальные заказы сохраняются должным образом. Ниже приведен тест.

```
...
[Fact]
public void Can_Checkout_And_Submit_Order() {
  // Организация - создание имитированного хранилища заказов
  Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
  // Организация - создание корзины с одним элементом
  Cart cart = new Cart();
  cart.AddItem(new Product(), 1);
  // Организация - создание экземпляра контроллера
  OrderController target = new OrderController(mock.Object, cart);
  // Действие - попытка перехода к оплате
  RedirectToActionResult result =
    target.Checkout(new Order()) as RedirectToActionResult;
  // Утверждение - проверка, что заказ был сохранен
  mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Once);
  // Утверждение - проверка, что метод перенаправляется на действие Completed
  Assert.Equal("Completed", result.ActionName);
}
...
```

Тестировать возможность идентификации допустимых сведений о доставке не нужно. Это автоматически обрабатывается средством привязки моделей с использованием атрибутов, примененных к свойствам класса `Order`.

Отображение сообщений об ошибках проверки достоверности

Для проверки пользовательских данных инфраструктура MVC будет использовать атрибуты проверки достоверности, примененные к классу `Order`. Тем не менее, чтобы отобразить сообщения о проблемах, понадобится внести небольшое изменение. Здесь задействована еще одна встроенная вспомогательная функция дескриптора, которая инспектирует состояние проверки достоверности данных, предоставленных пользователем, и добавляет предупреждающие сообщения для каждой обнаруженной проблемы. В листинге 10.18 демонстрируется добавление в файл `Checkout.cshtml` элемента HTML, который будет обрабатываться этой вспомогательной функцией дескриптора.

Листинг 10.18. Добавление области с итогами проверки достоверности в файле `Checkout.cshtml` из папки `Views/Order`

```
@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>
...
```

Благодаря такому простому изменению пользователю отображаются сообщения об ошибках проверки достоверности. Чтобы увидеть результат, понадобится посетить URL вида `/Order/Checkout` и попробовать перейти к оплате, не выбрав ни одного товара или не указав сведения о доставке (рис. 10.5). Вспомогательная функция дескриптора, генерирующая такие сообщения, является частью системы проверки достоверности моделей, которая подробно рассматривается в главе 27.

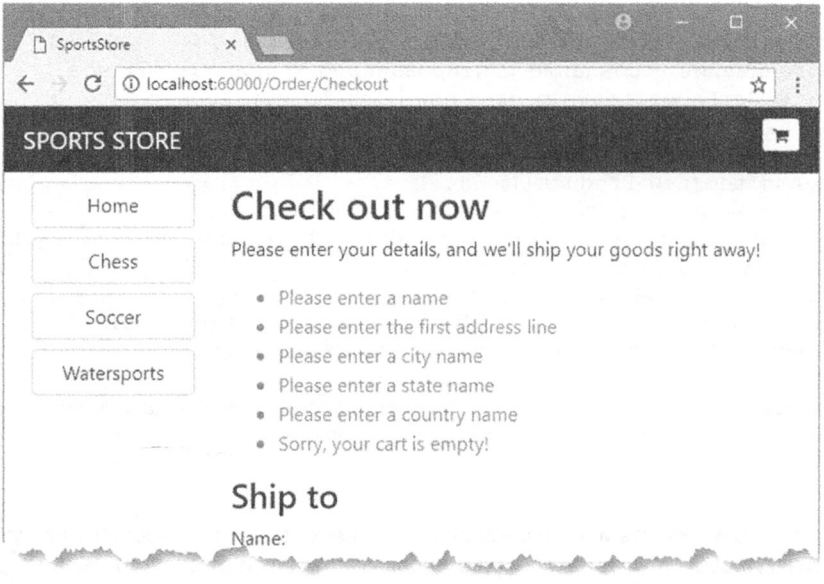

Рис. 10.5. Отображение сообщений об ошибках проверки достоверности

Совет. Данные, отправленные пользователем, перед проверкой посылаются серверу, что известно как *проверка достоверности на стороне сервера*, для которой инфраструктура MVC предлагает великолепную поддержку. Проблема с проверкой достоверности на стороне сервера заключается в том, что пользователю сообщается об ошибках лишь после того, как данные отправлены серверу и обработаны, а также сгенерирована результирующая страница — на занятом сервере все это может занять несколько секунд. По указанной причине проверка достоверности на стороне сервера обычно дополняется проверкой достоверности на стороне клиента, при которой введенные пользователем значения проверяются с помощью кода JavaScript до отправки серверу данных формы. Проверка достоверности на стороне клиента будет описана в главе 27.

Отображение итоговой страницы

Чтобы завершить реализацию процесса оплаты, необходимо создать представление, которое будет отображаться, когда браузер перенаправляется на действие `Completed` контроллера `Order`. Добавим в папку `Views/Order` файл представления Razor по имени `Completed.cshtml` и поместим в него разметку, приведенную в листинге 10.19.

Листинг 10.19. Содержимое файла `Completed.cshtml` из папки `Views/Order`

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

Для интеграции представления `Completed` в приложение никаких изменений в код вносить не придется, поскольку требуемые операторы уже были добавлены при определении метода действия `Completed()` в листинге 10.17. Теперь пользователь может проходить через весь процесс, начиная с выбора товаров и заканчивая переходом к оплате. При условии, что пользователь предоставил корректные сведения о доставке (и в корзине есть какие-то товары), после щелчка на кнопке Complete order он увидит итоговую страницу (рис. 10.6).

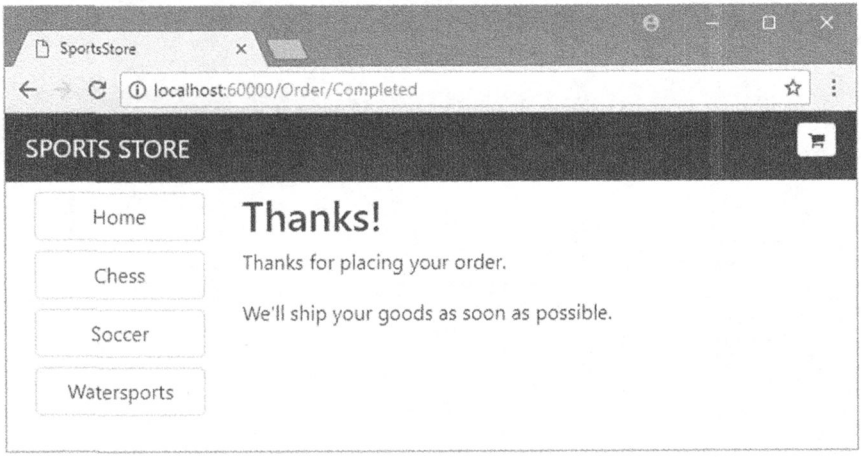

Рис. 10.6. Итоговая страница завершенного заказа

Резюме

Мы завершили все основные части приложения SportsStore, отвечающие за взаимодействие с пользователями. Конечно, до сайта Amazon приложению далеко, но в нем имеется каталог товаров с возможностью просмотра по категориям и страницам, аккуратная корзина для покупок и простой процесс оплаты.

Архитектура с хорошим разделением означает, что мы можем легко изменять поведение любой порции приложения, не беспокоясь о возникновении проблем или несовместимости где-либо в приложении. Например, мы могли бы изменить способ сохранения заказов, что никак бы не повлияло на корзину для покупок, каталог товаров или любую другую область приложения. В следующей главе мы добавим средства, необходимые для администрирования приложения SportsStore.

ГЛАВА 11

SportsStore: администрирование

В настоящей главе мы продолжим строить приложение SportsStore, чтобы предоставить администратору сайта способ управления заказами и товарами.

Управление заказами

В предыдущей главе была добавлена поддержка для получения заказов от пользователей и сохранения их в базе данных. В этой главе мы собираемся создать простой инструмент администрирования, который позволит просматривать полученные заказы и помечать их как отгруженные.

Расширение модели

Первым изменением, которое необходимо внести, является расширение модели, чтобы можно было фиксировать, какие заказы были отгружены. В листинге 11.1 показано добавление нового свойства в класс `Order`, который определен в файле `Order.cs` внутри папки `Models`.

Листинг 11.1. Добавление свойства в файле `Order.cs` из папки `Models`

```
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; }

    [BindNever]
    public bool Shipped { get; set; }

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

    [Required(ErrorMessage = "Please enter the first address line")]
                             // Введите первую строку адреса
```

```
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }

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

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

    [Required(ErrorMessage = "Please enter a country name")]
                            // Введите название страны
    public string Country { get; set; }
    public bool GiftWrap { get; set; }
  }
}
```

Такой итеративный подход к расширению и адаптации модели с целью поддержки различных средств типичен при разработке приложений MVC. В идеальном мире у нас была бы возможность полностью определить классы моделей в начале проекта и просто строить приложение на их основе, но так случается только в простейших проектах, а на практике следует ожидать итеративную разработку по мере понимания того, что требуется разрабатывать и развивать.

Миграции Entity Framework Core облегчают этот процесс, поскольку нам не приходится вручную удерживать схему базы данных в синхронизированном состоянии с классами моделей, создавая и запуская команды SQL. Обновим базу данных, чтобы отразить добавление свойства `Shipped` в класс `Order`, открыв окно командной строки или PowerShell, перейдя в папку проекта `SportsStore` (ту, что содержит файл `Startup.cs`) и выполнив следующую команду:

```
dotnet ef migrations add ShippedOrders
```

Миграция будет применена автоматически, когда приложение запустится и в классе `SeedData` произойдет обращение к методу `Migrate()`, предоставленному инфраструктурой Entity Framework Core.

Добавление действий и представления

Функциональность, требуемая для отображения и обновления набора заказов в базе данных, относительно проста, потому что она строится на основе средств и инфраструктуры, которые были созданы в предшествующих главах. В листинге 11.2 к контроллеру `Order` добавляются два метода действий.

Листинг 11.2. Добавление двух методов действий в файле `OrderController.cs` из папки `Controllers`

```
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 ViewResult List() =>
      View(repository.Orders.Where(o => !o.Shipped));

    [HttpPost]
    public IActionResult MarkShipped(int orderID) {
      Order order = repository.Orders
        .FirstOrDefault(o => o.OrderID == orderID);
      if (order != null) {
        order.Shipped = true;
        repository.SaveOrder(order);
      }
      return RedirectToAction(nameof(List));
    }

    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);
        return RedirectToAction(nameof(Completed));
      } else {
        return View(order);
      }
    }

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

Метод `List()` выбирает все объекты `Order` в хранилище, свойство `Shipped` которых имеет значение `false`, и передает их стандартному представлению. Метод действия `List()` будет использоваться для отображения администратору списка неотгруженных заказов.

Метод `MarkShipped()` будет получать запрос `POST`, указывающий идентификатор заказа, который применяется для извлечения соответствующего объекта `Order` из хранилища, чтобы установить его свойство `Shipped` в `true` и сохранить.

Для отображения списка неотгруженных заказов добавим в папку `Views/Order` файл представления Razor по имени `List.cshtml` и поместим в него разметку из листинга 11.3. Элемент `table` используется для отображения ряда деталей, включая сведения о приобретенных товарах.

Листинг 11.3. Содержимое файла `List.cshtml` из папки `Views/Order`

```
@model IEnumerable<Order>

@{
  ViewBag.Title = "Orders";
  Layout = "_AdminLayout";
}
@if (Model.Count() > 0) {
  <table class="table table-bordered table-striped">
    <tr><th>Name</th><th>Zip</th><th colspan="2">Details</th><th></th></tr>
    @foreach (Order o in Model) {
      <tr>
        <td>@o.Name</td><td>@o.Zip</td><th>Product</th><th>Quantity</th>
        <td>
          <form asp-action="MarkShipped" method="post">
            <input type="hidden" name="orderId" value="@o.OrderID" />
            <button type="submit" class="btn btn-sm btn-danger">
              Ship
            </button>
          </form>
        </td>
      </tr>
      @foreach (CartLine line in o.Lines) {
        <tr>
          <td colspan="2"></td>
          <td>@line.Product.Name</td><td>@line.Quantity</td>
          <td></td>
        </tr>
      }
    }
  </table>
} else {
  <div class="text-center">No Unshipped Orders</div>
}
```

Каждый заказ отображается с кнопкой Ship (Отгрузить), которая отправляет форму методу действия `MarkShipped()`. С помощью свойства `Layout` для представления `List` указана другая компоновка, которая переопределяет компоновку, заданную в файле `_ViewStart.cshtml`.

Для добавления компоновки создадим в папке `Views/Shared` файл по имени `_AdminLayout.cshtml` с применением шаблона элемента MVC View Layout Page (Страница компоновки представления MVC) и поместим в него разметку, показанную в листинге 11.4.

Листинг 11.4. Содержимое файла `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @RenderBody()
</body>
</html>
```

Чтобы просматривать и управлять заказами в приложении, запустим приложение, выберем некоторые товары и перейдем к оплате. Затем посетим URL вида `/Order/List`. Появится сводка по созданным заказам (рис. 11.1). Щелкнем на кнопке Ship; база данных обновится, а список ожидающих заказов будет пуст.

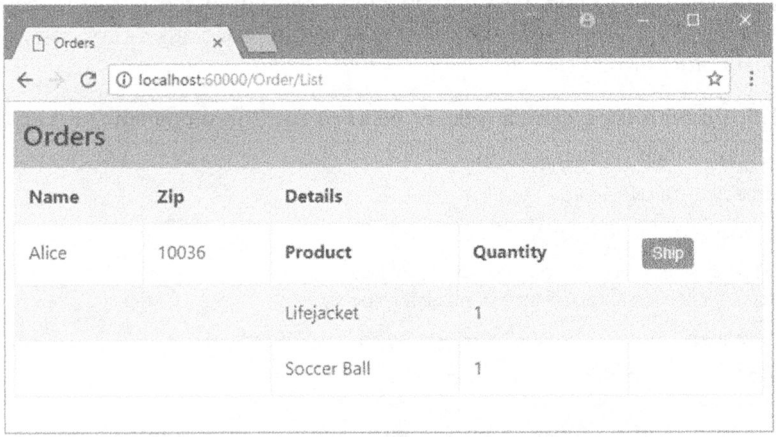

Рис. 11.1. Управление заказами

На заметку! В настоящий момент ничто не может воспрепятствовать запросу пользователем URL вида `/Order/List` и администрированию своего заказа. В главе 12 объясняется, как ограничить доступ к методам действий.

Добавление средств управления каталогом

Соглашение для управления более сложными коллекциями элементов предусматривает предоставление пользователю страниц двух типов: страницы *списка* и страницы *редактирования* (рис. 11.2).

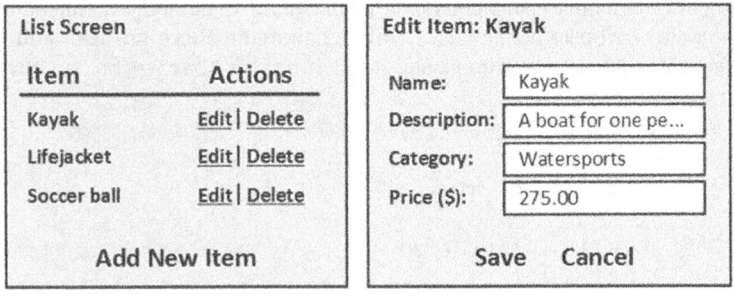

Рис. 11.2. Эскиз пользовательского интерфейса CRUD для каталога товаров

Вместе эти страницы позволяют пользователю создавать, читать, обновлять и удалять (create, read, update, delete — CRUD) элементы в коллекции. Такие действия называются *операциями CRUD*. Разработчики нуждаются в реализации операций CRUD настолько часто, что средство формирования шаблонов в Visual Studio предлагает сценарии для создания контроллеров CRUD с заранее определенными методами действий (включение средства формирования шаблонов рассматривалось в главе 8). Но, как и со всеми шаблонами Visual Studio, я считаю, что изучать возможности ASP.NET Core MVC лучше напрямую.

Создание контроллера CRUD

Начнем с создания отдельного контроллера для управления каталогом товаров. Добавим в папку `Controllers` файл класса по имени `AdminController.cs` с кодом, приведенным в листинге 11.5.

Листинг 11.5. Содержимое файла `AdminController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

    public ViewResult Index() => View(repository.Products);
  }
}
```

В конструкторе контроллера объявлена зависимость от интерфейса `IProductRepository`, которая будет распознаваться при создании экземпляров. В классе контроллера определен единственный метод действия `Index()`, который вызывает метод `View()`, чтобы выбрать стандартное представление для действия, и передает ему в качестве модели представления набор товаров из базы данных.

Модульное тестирование: метод действия `Index()`

Нас интересует поведение метода действия `Index()` в контроллере `Admin`, которое заключается в корректном возвращении объектов `Product` из хранилища. Протестировать его можно за счет создания имитированной реализации хранилища и сравнения тестовых данных с данными, которые возвращает метод действия. Ниже показан код модульного теста, помещенный в новый файл по имени `AdminControllerTests.cs` внутри проекта `SportsStore.Tests`.

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;
```

```
namespace SportsStore.Tests {
  public class AdminControllerTests {
    [Fact]
    public void Index_Contains_All_Products() {
      // Организация - создание имитированного хранилища
      Mock<IProductRepository> mock = new Mock<IProductRepository>();
      mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
      }).AsQueryable<Product>());
      // Организация - создание контроллера
      AdminController target = new AdminController(mock.Object);
      // Действие
      Product[] result
        = GetViewModel<IEnumerable<Product>>(target.Index())?.ToArray();
      // Утверждение
      Assert.Equal(3, result.Length);
      Assert.Equal("P1", result[0].Name);
      Assert.Equal("P2", result[1].Name);
      Assert.Equal("P3", result[2].Name);
    }
    private T GetViewModel<T>(IActionResult result) where T : class {
      return (result as ViewResult)?.ViewData.Model as T;
    }
  }
}
```

В тест был добавлен метод `GetViewModel()` для распаковки результата, возвращаемого методом действия, и получения данных модели представления. Далее в главе будут реализованы дополнительные тесты, которые используют этот метод.

Реализация представления списка

Следующим шагом будет добавление представления для метода действия `Index()` контроллера `Admin`. Создадим папку `Views/Admin` и добавим в нее файл представления Razor по имени `Index.cshtml` с содержимым, приведенным в листинге 11.6.

Листинг 11.6. Содержимое файла `Index.cshtml` из папки `Views/Admin`

```
@model IEnumerable<Product>

@{
  ViewBag.Title = "All Products";
  Layout = "_AdminLayout";
}

<table class="table table-striped table-bordered table-sm">
  <tr>
    <th class="text-right">ID</th>
    <th>Name</th>
    <th class="text-right">Price</th>
    <th class="text-center">Actions</th>
  </tr>
```

```
  @foreach (var item in Model) {
    <tr>
      <td class="text-right">@item.ProductID</td>
      <td>@item.Name</td>
      <td class="text-right">@item.Price.ToString("c")</td>
      <td class="text-center">
        <form asp-action="Delete" method="post">
          <a asp-action="Edit" class="btn btn-sm btn-warning"
             asp-route-productId="@item.ProductID">
            Edit
          </a>
          <input type="hidden" name="ProductID" value="@item.ProductID" />
          <button type="submit" class="btn btn-danger btn-sm">
            Delete
          </button>
        </form>
      </td>
    </tr>
  }
</table>
<div class="text-center">
  <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>
```

Представление содержит таблицу, в которой для каждого товара предусмотрена строка с ячейками, содержащими наименование и цену товара. Кроме того, в каждой строке присутствуют кнопки, которые позволят редактировать сведения о товаре и удалять его, отправляя запросы к действиям `Edit` и `Delete`. В дополнение к таблице имеется кнопка Add Product (Добавить товар), нацеленная на действие `Create`. Мы добавим действия `Edit`, `Delete` и `Create` в последующих разделах, а пока можно посмотреть, как отображаются товары, запустив приложение и запросив URL вида `/Admin/Index` (рис. 11.3).

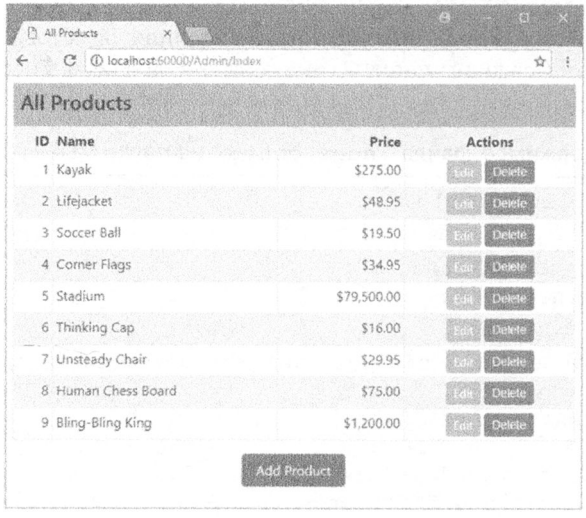

Рис. 11.3. Отображение списка товаров

Совет. В листинге 11.6 кнопка Edit (Редактировать) находится внутри элемента `form`, так что две кнопки располагаются рядом благодаря интервалу, примененному Bootstrap. Кнопка Edit будет посылать серверу HTTP-запрос `GET` для получения текущих сведений о товаре; это не требует элемента `form`. Однако поскольку кнопка Delete (Удалить) будет вносить изменение в состояние приложения, необходимо использовать HTTP-запрос `POST`, который требует элемента `form`.

Редактирование сведений о товарах

Чтобы предоставить средства создания и обновления, мы добавим страницу редактирования сведений о товаре, подобную показанной на рис. 11.2. Задача состоит из двух частей:

- отобразить страницу, которая позволит администратору изменять значения для свойств товара;
- добавить метод действия, который обработает внесенные изменения, когда они будут отправлены.

Создание метода действия `Edit()`

В листинге 11.7 приведен код метода действия `Edit()`, добавленного в контроллер `Admin`, который будет получать HTTP-запрос, отправляемый браузером, когда пользователь щелкает на кнопке Edit.

Листинг 11.7. Добавление метода действия `Edit()` в файле `AdminController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));
  }
}
```

Этот простой метод ищет товар с идентификатором, соответствующим значению параметра `productId`, и передает его как объект модели представления методу `View()`.

Модульное тестирование: метод действия Edit()

В методе действия Edit() нам необходимо протестировать две линии поведения. Первая заключается в том, что мы получаем запрашиваемый товар, когда предоставляем допустимое значение идентификатора, чтобы удостовериться в редактировании ожидаемого товара. Вторая проверяемая линия поведения связана с тем, что мы не должны получать товар при запросе значения идентификатора, отсутствующего в хранилище. Ниже показаны тестовые методы, добавленные в файл класса AdminControllerTests.cs.

```
...
[Fact]
public void Can_Edit_Product() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
  }).AsQueryable<Product>());
  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие
  Product p1 = GetViewModel<Product>(target.Edit(1));
  Product p2 = GetViewModel<Product>(target.Edit(2));
  Product p3 = GetViewModel<Product>(target.Edit(3));

  // Утверждение
  Assert.Equal(1, p1.ProductID);
  Assert.Equal(2, p2.ProductID);
  Assert.Equal(3, p3.ProductID);
}

[Fact]
public void Cannot_Edit_Nonexistent_Product() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
  mock.Setup(m => m.Products).Returns((new Product[] {
    new Product {ProductID = 1, Name = "P1"},
    new Product {ProductID = 2, Name = "P2"},
    new Product {ProductID = 3, Name = "P3"},
  }).AsQueryable<Product>());

  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие
  Product result = GetViewModel<Product>(target.Edit(4));

  // Утверждение
  Assert.Null(result);
}
...
```

Создание представления редактирования

Теперь, располагая методом действия, можно создать представление для отображения. Добавим в папку `Views/Admin` файл представления Razor по имени `Edit.cshtml` и поместим в него разметку, приведенную в листинге 11.8.

Листинг 11.8. Содержимое файла `Edit.cshtml` из папки `Views/Admin`

```
@model Product

@{
  ViewBag.Title = "Edit Product";
  Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
  <input type="hidden" asp-for="ProductID" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" class="form-control"></textarea>
  </div>
  <div class="form-group">
    <label asp-for="Category"></label>
    <input asp-for="Category" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Price"></label>
    <input asp-for="Price" class="form-control" />
  </div>
  <div class="text-center">
    <button class="btn btn-primary" type="submit">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

В представлении имеется форма HTML, большая часть содержимого которой генерируется посредством вспомогательных функций дескрипторов, включая установку целей для элементов `form` и `a`, установку содержимого элементов `label` и выдачу атрибутов `name`, `id` и `value` для элементов `input` и `textarea`.

Чтобы увидеть HTML-разметку, генерируемую представлением, запустим приложение, перейдем на URL типа `/Admin/Index` и щелкнем на кнопке Edit для одного из товаров (рис. 11.4).

Совет. Скрытый элемент `input` для свойства `ProductID` применяется ради простоты. Значение `ProductID` генерируется базой данных как первичный ключ, когда новый объект сохраняется инфраструктурой Entity Framework Core, и его безопасное изменение может оказаться сложным процессом. Для большинства приложений проще всего предотвратить изменение значения со стороны пользователя.

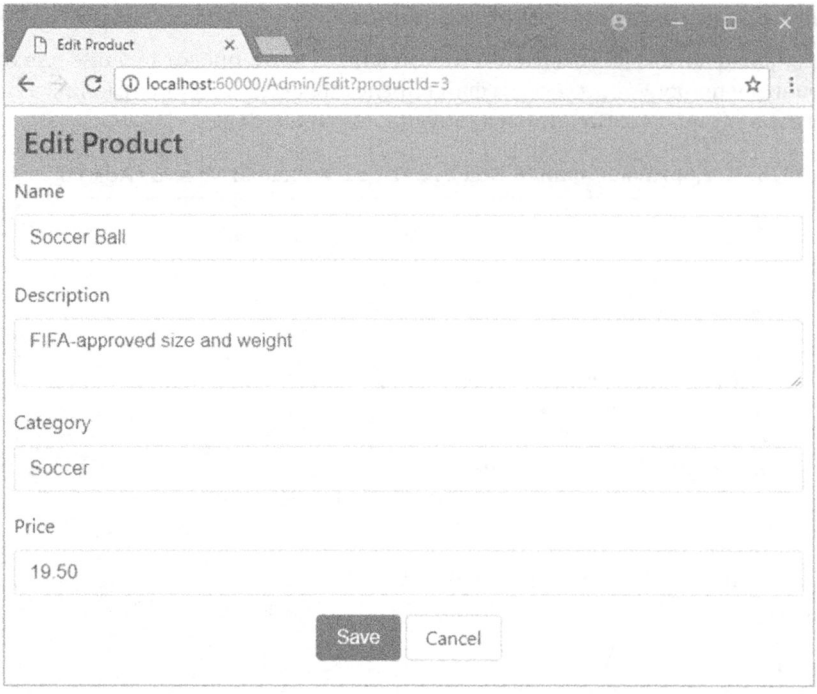

Рис. 11.4. Отображение сведений о товаре для редактирования

Обновление хранилища товаров

Прежде чем можно будет обрабатывать результаты редактирования, хранилище товаров понадобится расширить, добавив возможность сохранения изменений. Первым делом необходимо добавить к интерфейсу `IProductRepository` новый метод (листинг 11.9).

Листинг 11.9. Добавление метода в файл `IProductRespository.cs` из папки `Models`

```
using System.Linq;
namespace SportsStore.Models {
  public interface IProductRepository {
    IQueryable<Product> Products { get; }
    void SaveProduct(Product product);
  }
}
```

Затем можно добавить новый метод к реализации хранилища с помощью Entity Framework Core, которая определена в файле `EFProductRepository.cs` (листинг 11.10).

Листинг 11.10. Реализация нового метода в файле `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Product> Products => context.Products;
    public void SaveProduct(Product product) {
      if (product.ProductID == 0) {
        context.Products.Add(product);
      } else {
        Product dbEntry = context.Products
          .FirstOrDefault(p => p.ProductID == product.ProductID);
        if (dbEntry != null) {
          dbEntry.Name = product.Name;
          dbEntry.Description = product.Description;
          dbEntry.Price = product.Price;
          dbEntry.Category = product.Category;
        }
      }
      context.SaveChanges();
    }
  }
}
```

Реализация метода `SaveChanges()` добавляет товар в хранилище, если значение `ProductID` равно 0; в противном случае применяются изменения к существующей записи в базе данных.

Мы не хотим здесь вдаваться в детали инфраструктуры Entity Framework Core, поскольку, как упоминалось ранее, это отдельная крупная тема, к тому же она не является частью ASP.NET Core MVC. Тем не менее, кое-какой код в методе `SaveProduct()` оказывает влияние на проектное решение, положенное в основу приложения MVC.

Нам известно, что обновление должно выполняться, когда получен параметр `Product`, который имеет ненулевое значение `ProductID`. Задача решается путем извлечения из хранилища объекта `Product` с тем же самым значением `ProductID` и обновления всех его свойств, чтобы они соответствовали значениям свойств объекта, переданного в качестве параметра.

Причина таких действий в том, что инфраструктура Entity Framework Core отслеживает объекты, которые она создает из базы данных. Объект, переданный методу `SaveChanges()`, создается системой привязки моделей MVC, т.е. инфраструктура Entity Framework Core ничего не знает о новом объекте `Product`, и она не будет применять обновление к базе данных, когда объект `Product` модифицирован. Существует множество способов решения указанной проблемы, но мы принимаем самый простой из них, предполагающий поиск соответствующего объекта, о котором известно инфраструктуре Entity Framework Core, и его явное обновление.

Добавление нового метода в интерфейс IProductRepository нарушает работу класса имитированного хранилища FakeProductRepository, который был создан в главе 8. Имитированное хранилище использовалось в целях быстрого старта процесса разработки и демонстрации возможности применения служб для гладкой замены реализаций интерфейса, не изменяя компоненты, которые на них опираются. Имитированное хранилище больше не понадобится. В листинге 11.11 видно, что интерфейс IProductRepository удален из объявления класса, поэтому продолжать модификацию класса по мере добавления функций хранилища не придется.

Листинг 11.11. Удаление интерфейса в файле FakeProductRepository.cs из папки Models

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class FakeProductRepository /* : IProductRepository */ {

    public IQueryable<Product> Products => new List<Product> {
      new Product { Name = "Football", Price = 25 },
      new Product { Name = "Surf board", Price = 179 },
      new Product { Name = "Running shoes", Price = 95 }
    }.AsQueryable<Product>();
  }
}
```

Обработка запросов POST в методе действия Edit()

К настоящему моменту все готово для реализации в контроллере Admin перегруженной версии метода действия Edit(), которая будет обрабатывать запросы POST, инициируемые по щелчку администратором на кнопке Save (Сохранить). Код нового метода приведен в листинге 11.12.

Листинг 11.12. Определение метода действия в файле AdminController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
```

```
    public IActionResult Edit(Product product) {
      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
  }
}
```

Мы выясняем, смог ли процесс привязки модели проверить достоверность отправленных пользователем данных, для чего читаем значение свойства `ModelState.IsValid`. Если здесь все в порядке, тогда мы сохраняем изменения в хранилище и направляем пользователя на действие `Index`, так что он увидит модифицированный список товаров. При наличии какой-нибудь проблемы с данными мы снова визуализируем стандартное представление, чтобы пользователь получил возможность внести корректировки.

После сохранения изменений в хранилище сообщение сохраняется с использованием *средства* `TempData`, которое является частью средства состояния сеанса ASP.NET Core. Это словарь пар "ключ/значение", похожий на применяемые ранее средства данных сеанса и `ViewBag`. Основное отличие объекта `TempData` от данных сеанса в том, что он хранится до тех пор, пока не будет прочитан.

В такой ситуации использовать `ViewBag` невозможно, потому что объект `ViewBag` передает данные между контроллером и представлением, и он не может удерживать данные дольше, чем длится текущий HTTP-запрос. Когда редактирование успешно, браузер перенаправляется на новый URL, поэтому данные `ViewBag` утрачиваются. Мы могли бы прибегнуть к средству данных сеанса, но тогда сообщение хранилось бы вплоть до его явного удаления, чего делать бы не хотелось.

Таким образом, объект `TempData` подходит как нельзя лучше. Данные ограничиваются сеансом одного пользователя (пользователи не видят объекты `TempData` друг друга) и хранятся достаточно долго, чтобы быть прочитанными. Мы будем читать данные в представлении, которое визуализируется методом действия, куда перенаправлялся пользователь, и определяется в следующем разделе.

Модульное тестирование: метод действия `Edit()`, обрабатывающий запросы `POST`

В методе действия `Edit()`, обрабатывающем запросы `POST`, мы должны удостовериться, что хранилищу товаров для сохранения передаются допустимые обновления объекта `Product`, полученного в качестве аргумента метода. Кроме того, необходимо проверить, что недопустимые обновления (т.е. содержащие ошибки проверки достоверности модели) в хранилище не передаются. Ниже приведены тестовые методы, которые добавлены в файл `AdminControllerTests.cs`.

```
...
[Fact]
public void Can_Save_Valid_Changes() {
  // Организация - создание имитированного хранилища
  Mock<IProductRepository> mock = new Mock<IProductRepository>();
```

```
    // Организация - создание имитированных временных данных
    Mock<ITempDataDictionary> tempData = new Mock<ITempDataDictionary>();
    // Организация - создание контроллера
    AdminController target = new AdminController(mock.Object) {
      TempData = tempData.Object
    };
    // Организация - создание товара
    Product product = new Product { Name = "Test" };

    // Действие - попытка сохранить товар
    IActionResult result = target.Edit(product);

    // Утверждение - проверка того, что к хранилищу было произведено обращение
    mock.Verify(m => m.SaveProduct(product));
    // Утверждение - проверка, что типом результата является перенаправление
    Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Index", (result as RedirectToActionResult).ActionName);
  }

  [Fact]
  public void Cannot_Save_Invalid_Changes() {
    // Организация - создание имитированного хранилища
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    // Организация - создание контроллера
    AdminController target = new AdminController(mock.Object);
    // Организация - создание товара
    Product product = new Product { Name = "Test" };
    // Организация - добавление ошибки в состояние модели
    target.ModelState.AddModelError("error", "error");

    // Действие - попытка сохранить товар
    IActionResult result = target.Edit(product);

    // Утверждение - проверка того, что к хранилищу было произведено обращение
    mock.Verify(m => m.SaveProduct(It.IsAny<Product>()), Times.Never());
    // Утверждение - проверка типа результата метода
    Assert.IsType<ViewResult>(result);
  }
  ...
```

Отображение подтверждающего сообщения

Мы будем иметь дело с сообщением, сохраненным с помощью `TempData`, в файле компоновки `_AdminLayout.cshtml` (листинг 11.13). За счет обработки сообщения в компоновке мы можем создавать сообщения в любом представлении, которое применяет данную компоновку, без необходимости в создании дополнительных выражений Razor.

Листинг 11.13. Обработка сообщения `TempData` в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Совет. Преимущество работы с сообщением внутри компоновки заключается в том, что пользователи будут видеть его на любой странице, визуализированной после сохранения изменений. В данный момент мы возвращаем его списку товаров, но рабочий поток можно изменить с целью визуализации какого-то другого представления, и пользователи будут по-прежнему видеть сообщение (при условии, что следующее представление использует ту же самую компоновку).

Теперь мы располагаем всеми фрагментами для редактирования сведений о товарах. Чтобы увидеть, как все они работают, запустим приложение, перейдем на URL вида `/Admin/Index`, щелкнем на кнопке Edit и внесем изменение. Затем щелкнем на кнопке Save. Произойдет перенаправление на `/Admin/Index` и отобразится сообщение из `TempData` (рис. 11.5). Если перезагрузить страницу со списком товаров, то сообщение исчезнет, поскольку после чтения объект `TempData` удаляется. Подход очень удобен, т.к. не приходится иметь дело со старыми сообщениями.

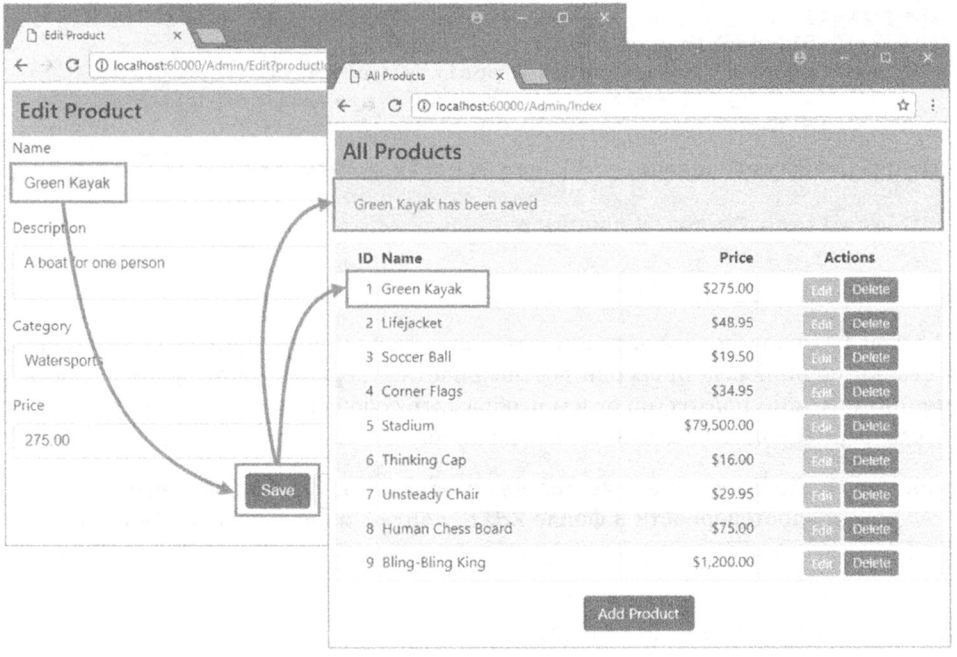

Рис. 11.5. Редактирование сведений о товаре и отображение сообщения из `TempData`

Добавление проверки достоверности модели

Мы добрались до точки, когда к классам модели необходимо добавить правила проверки достоверности. Пока что администратор может вводить отрицательные значения для цен или оставлять описания пустыми — и приложение SportsStore благополучно сохранит эти данные в базе. Смогут ли недопустимые данные успешно сохраниться, зависит от того, удовлетворяют ли они ограничениям в таблицах SQL, созданных инфраструктурой Entity Framework Core, и для большинства приложений таких мер безопасности будет недостаточно. Чтобы защититься от недопустимых значений данных, свойства класса `Product` декорируются с помощью атрибутов, как делалось в классе `Order` из главы 10 (листинг 11.14).

Листинг 11.14. Применение атрибутов проверки достоверности в файле `Product.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class Product {
    public int ProductID { get; set; }

    [Required(ErrorMessage = "Please enter a product name")]
                              // Введите наименование товара
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter a description")]
                             // Введите описание
    public string Description { get; set; }

    [Required]
    [Range(0.01, double.MaxValue,
      ErrorMessage = "Please enter a positive price")]
                    // Введите положительное значение для цены
    public decimal Price { get; set; }

    [Required(ErrorMessage = "Please specify a category")]
                              // Укажите категорию
    public string Category { get; set; }
  }
}
```

В главе 10 использовалась вспомогательная функция дескриптора для отображения сводки по ошибкам проверки достоверности в верхней части формы. Здесь мы применим похожий подход, но будем отображать сообщения об ошибках рядом с элементами формы в представлении `Edit` (листинг 11.15).

Листинг 11.15. Добавление элементов для отображения ошибок проверки достоверности в файле `Edit.cshtml` из папки `Views/Admin`

```
@model Product

@{
  ViewBag.Title = "Edit Product";
  Layout = "_AdminLayout";
}
```

```
<form asp-action="Edit" method="post">
  <input type="hidden" asp-for="ProductID" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <div><span asp-validation-for="Name" class="text-danger"></span></div>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Description"></label>
    <div><span asp-validation-for="Description" class="text-danger"></span>
    </div>
    <textarea asp-for="Description" class="form-control"></textarea>
  </div>
  <div class="form-group">
    <label asp-for="Category"></label>
    <div><span asp-validation-for="Category" class="text-danger"></span>
    </div>
    <input asp-for="Category" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Price"></label>
    <div><span asp-validation-for="Price" class="text-danger"></span>
    </div>
    <input asp-for="Price" class="form-control" />
  </div>
  <div class="text-center">
    <button class="btn btn-primary" type="submit">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Когда атрибут `asp-validation-for` применяется к элементу `span`, он использует вспомогательную функцию дескриптора, которая добавляет сообщение об ошибке проверки достоверности для указанного свойства, если при проверке возникли какие-то проблемы.

Вспомогательные функции дескрипторов будут вставлять сообщение об ошибке в элемент `span` и добавлять элемент в класс `input-validation-error`, который позволит легко применять стили CSS к элементам с сообщениями об ошибках (листинг 11.16).

Листинг 11.16. Добавление стиля CSS в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error { border-color: red; background-color: #fee ; }
  </style>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Определенный здесь стиль CSS выбирает элементы, которые являются членами класса `input-validation-error`, и устанавливает для них границу красного цвета и фон.

Совет. Явная установка стилей, когда используется библиотека CSS наподобие Bootstrap, может привести к несоответствиям в случае применения тем содержимого. В главе 27 будет продемонстрирован альтернативный подход, при котором для применения классов Bootstrap к элементам с сообщениями об ошибках проверки достоверности используется код JavaScript, сохраняя все в согласованном состоянии.

Вспомогательные функции дескрипторов для сообщений проверки достоверности можно применять где угодно в представлении, но по соглашению (что вполне разумно) принято размещать их поближе к проблемному элементу, чтобы ввести пользователя в курс дела. На рис. 11.6 показано, как выглядят сообщения об ошибках проверки достоверности и отображаемые подсказки, для чего нужно запустить приложение, отредактировать сведения о товаре и отправить недопустимые данные.

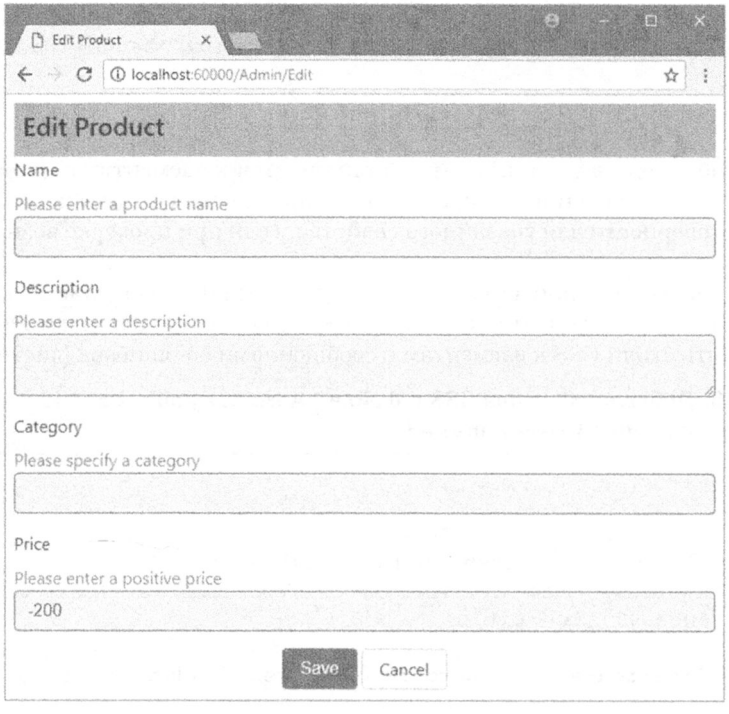

Рис. 11.6. Проверка достоверности данных при редактировании сведений о товаре

Включение проверки достоверности на стороне клиента

В текущий момент проверка достоверности данных применяется, только когда пользователь-администратор отправляет результаты редактирования серверу, но большинство пользователей ожидают немедленного отклика при наличии проблем с введенными данными. Именно потому разработчики часто предпочитают выполнять *проверку достоверности на стороне клиента*, при которой данные проверяются в браузере с использованием JavaScript. Приложения MVC могут выполнять проверку достоверности на стороне клиента на основе аннотаций данных, применяемых к классу модели предметной области.

Прежде всего, понадобится добавить библиотеки JavaScript, которые предоставят приложению средство проверки достоверности на стороне клиента, что делается в файле `bower.json` (листинг 11.17).

Листинг 11.17. Добавление пакетов JavaScript в файле `bower.json` из папки `SportsStore`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "fontawesome": "4.7.0",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Проверка достоверности на стороне клиента построена на основе популярной библиотеки jQuery, которая упрощает работу с API-интерфейсом DOM браузера. Следующий шаг связан с добавлением файлов JavaScript в компоновку, чтобы они загружались, когда используются средства администрирования приложения SportsStore (листинг 11.18).

Листинг 11.18. Добавление библиотек проверки достоверности в файле `_AdminLayout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error { border-color: red; background-color: #fee ; }
  </style>
  <script src="/lib/jquery/dist/jquery.min.js"></script>
  <script src="/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script
src="/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
  </script>
</head>
```

```
<body class="m-1 p-1">
  <div class="bg-info p-2"><h4>@ViewBag.Title</h4></div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>
```

Включение проверки достоверности на стороне клиента не приводит к каким-то визуальным изменениям. Но ограничения, которые указаны с помощью атрибутов, примененных к классу модели C#, вступают в силу на уровне браузера, предотвращая отправку пользователем формы с недопустимыми данными и обеспечивая немедленный отклик при наличии проблемы. За дополнительными сведениями обращайтесь в главу 27.

Создание новых товаров

Далее мы реализуем метод действия `Create()`, который указан для кнопки Add Product на главной странице со списком товаров. Он позволит администратору добавлять новые элементы в каталог товаров. Добавление возможности создания новых товаров требует одного небольшого дополнения в приложении, что является великолепной демонстрацией мощи и гибкости хорошо структурированного приложения MVC. Для начала добавим в контроллер `Admin` метод `Create()`, как показано в листинге 11.19.

Листинг 11.19. Добавление метода действия `Create()` в файле `AdminController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
  public class AdminController : Controller {
    private IProductRepository repository;
    public AdminController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult Index() => View(repository.Products);
    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
    public IActionResult Edit(Product product) {
      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
```

```
    public ViewResult Create() => View("Edit", new Product());
  }
}
```

Метод `Create()` не визуализирует свое стандартное представление. Взамен он указывает, что должно использоваться представление `Edit`. Применение в методе действия представления, которое обычно связано с другим методом действия, вполне допустимо. В рассматриваемом случае мы указываем в качестве модели представления новый объект `Product`, так что представление `Edit` заполняется пустыми полями.

На заметку! Модульный тест для метода действия `Create()` не добавляется. Он позволил бы проверить только способность обработки инфраструктурой ASP.NET Core MVC результата, возвращаемого методом действия — то, что мы считаем само собой разумеющимся. (Обычно тесты для функциональных средств инфраструктуры не пишутся, если только нет подозрения о наличии дефекта.)

Это единственное изменение, которое потребовалось внести, поскольку метод действия `Edit()` уже настроен на получение объектов `Product` от системы привязки моделей и на их сохранение в базе данных. Чтобы протестировать имеющуюся функциональность, запустим приложение, перейдем на URL вида `/Admin/Index`, щелкнем на кнопке Add Product, заполним форму и отправим ее. Информация, введенная на форме, будет использоваться для создания нового товара в базе данных, который затем появится в списке (рис. 11.7).

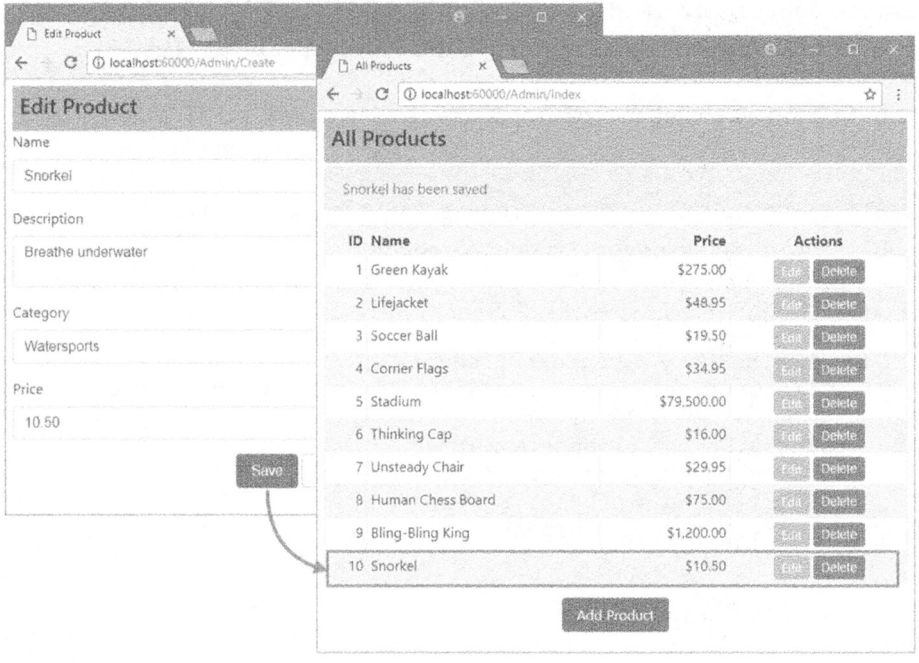

Рис. 11.7. Добавление нового товара в каталог

Удаление товаров

Обеспечить поддержку удаления элементов из каталога довольно просто. Для начала добавим в интерфейс `IProductRepository` новый метод, как показано в листинге 11.20.

Листинг 11.20. Добавление метода для удаления товаров в файле `IProductRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {

  public interface IProductRepository {

    IQueryable<Product> Products { get; }

    void SaveProduct(Product product);

    Product DeleteProduct(int productID);
  }
}
```

Затем реализуем метод `DeleteProduct()` в классе хранилища Entity Framework Core, т.е. `EFProductRepository` (листинг 11.21).

Листинг 11.21. Реализация поддержки удаления в файле `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }

    public IQueryable<Product> Products => context.Products;

    public void SaveProduct(Product product) {
      if (product.ProductID == 0) {
        context.Products.Add(product);
      } else {
        Product dbEntry = context.Products
          .FirstOrDefault(p => p.ProductID == product.ProductID);
        if (dbEntry != null) {
          dbEntry.Name = product.Name;
          dbEntry.Description = product.Description;
          dbEntry.Price = product.Price;
          dbEntry.Category = product.Category;
        }
      }
      context.SaveChanges();
    }
```

```
    public Product DeleteProduct(int productID) {
      Product dbEntry = context.Products
        .FirstOrDefault(p => p.ProductID == productID);
      if (dbEntry != null) {
        context.Products.Remove(dbEntry);
        context.SaveChanges();
      }
      return dbEntry;
    }
  }
}
```

Финальный шаг связан с реализацией метода действия `Delete()` в контроллере `Admin`. Метод действия `Delete()` должен поддерживать только запросы `POST`, потому что удаление объектов не является идемпотентной операцией. Как будет показано в главе 16, браузеры и кеши вольны выдавать запросы `GET` без явного согласия пользователя, поэтому мы должны проявить осторожность, чтобы избежать внесения изменений как следствия запросов `GET`. Код нового метода действия приведен в листинге 11.22.

Листинг 11.22. Добавление метода действия `Delete()` в файле `AdminController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

    public ViewResult Index() => View(repository.Products);

    public ViewResult Edit(int productId) =>
      View(repository.Products
        .FirstOrDefault(p => p.ProductID == productId));

    [HttpPost]
    public IActionResult Edit(Product product) {

      if (ModelState.IsValid) {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
      } else {
        // Что-то не так со значениями данных
        return View(product);
      }
    }
```

```
    public IActionResult Create() => View("Edit", new Product());

    [HttpPost]
    public IActionResult Delete(int productId) {
      Product deletedProduct = repository.DeleteProduct(productId);
      if (deletedProduct != null) {
        TempData["message"] = $"{deletedProduct.Name} was deleted";
      }
      return RedirectToAction("Index");
    }
  }
}
```

Модульное тестирование: удаление товаров

Нам нужно протестировать основное поведение метода действия `Delete()`, которое заключается в том, что при передаче в качестве параметра допустимого идентификатора `ProductID` метод действия должен вызвать метод `DeleteProduct()` хранилища и передать ему корректное значение `ProductID` удаляемого товара. Вот тест, добавленный в файл `AdminControllerTests.cs`:

```
...
[Fact]
public void Can_Delete_Valid_Products() {
  // Организация - создание объекта Product
  Product prod = new Product { ProductID = 2, Name = "Test" };

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

  // Организация - создание контроллера
  AdminController target = new AdminController(mock.Object);

  // Действие - удаление товара
  target.Delete(prod.ProductID);

  // Утверждение - проверка того, что был вызван метод удаления
  // в хранилище с корректным объектом Product
  mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
...
```

Чтобы увидеть средство удаления в работе, запустим приложение, перейдем на URL вида `/Admin/Index` и щелкнем на одной из кнопок Delete (Удалить) внутри страницы со списком товаров (рис. 11.8). На рисунке можно заметить, что с помощью переменной `TempData` отображается сообщение об удалении товара из каталога.

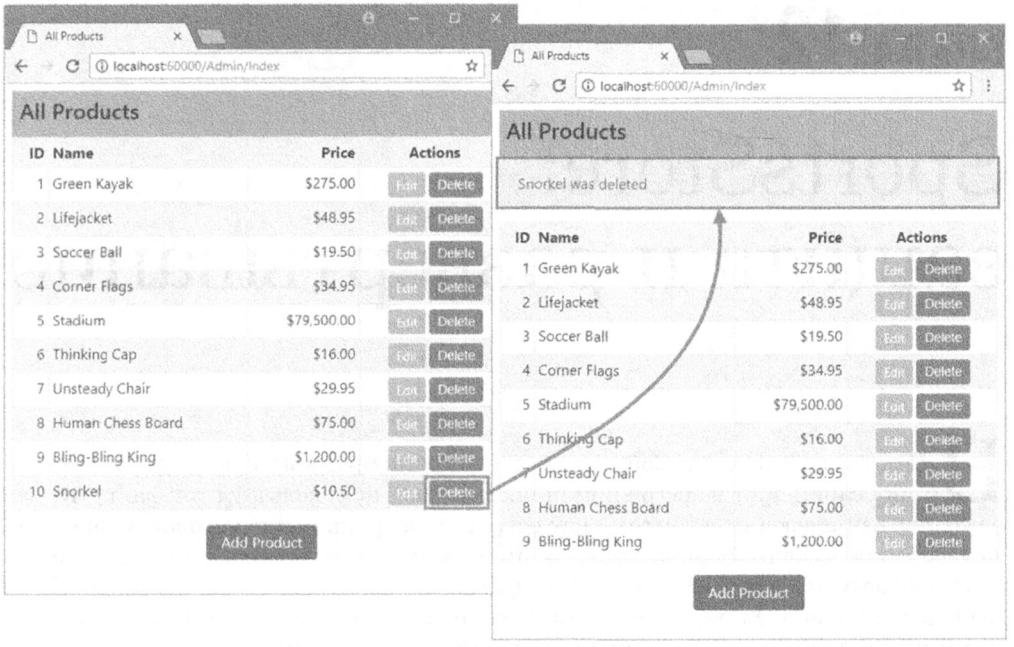

Рис. 11.8. Удаление товара из каталога

На заметку! При попытке удалить товар, для которого ранее был создан заказ, возникнет ошибка. Когда объект `Order` сохраняется в базе данных, он превращается в запись внутри таблицы базы данных, которая содержит ссылку на связанный с ним объект `Product`, что известно как отношение внешнего ключа. Это означает, что по умолчанию база данных не позволит удалить объект `Product`, если для него был создан объект `Order`, поскольку такое действие внесло бы несогласованность в базу данных. Существует несколько способов решения указанной проблемы, включая удаление объектов `Order` при удалении объекта `Product`, к которому они относятся, либо изменение отношения между объектами `Product` и `Order`. Подробные сведения можно найти в документации Entity Framework Core.

Резюме

В главе были введены средства администрирования и показано, как реализовать операции CRUD, которые позволяют администратору создавать, читать, обновлять и удалять товары из хранилища и помечать заказы как отгруженные. В следующей главе мы продемонстрируем способ защиты административных функций, чтобы они не были доступными абсолютно всем пользователям, и развернем приложение SportsStore в производственной среде.