ГЛАВА 9

SportsStore: навигация

В этой главе мы продолжим построение примера приложения SportsStore. В предыдущей главе была добавлена поддержка для навигации по приложению и для начала создания корзины для покупок.

Добавление навигационных элементов управления

Приложение SportsStore станет намного удобнее, если пользователи получат возможность навигации среди товаров по категории. Это будет делаться в три этапа.

- Расширение метода действия `List()` в классе `ProductController`, чтобы он был способен фильтровать объекты `Product` в хранилище.
- Пересмотр и расширение схемы URL.
- Создание списка 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`

```
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`

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

В этот метод действия внесены три изменения. Первое — добавлен новый параметр по имени `category`. Он применяется вторым изменением, которое представляет собой расширение 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}
  });
  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 также 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}
  });
  // 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, запустите приложение и выберите категорию с помощью показанной ниже строки запроса, заменив 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
```

Вы увидите только товары из категории `Soccer` (рис. 9.1).

Очевидно, что пользователи не 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.

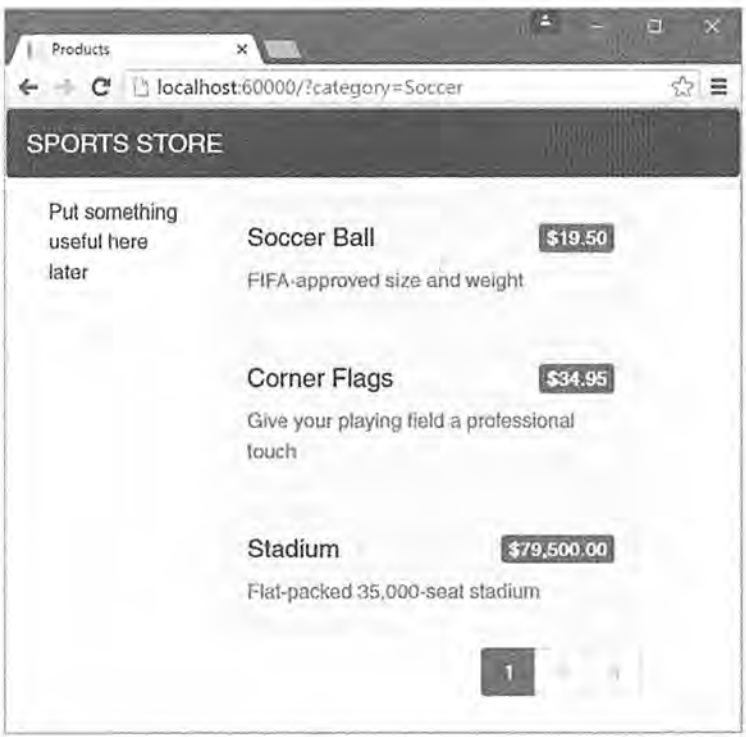

Рис. 9.1. Использование строки запроса для фильтрации по категории

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}
  });

  // 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);
}
...
```

Этот тест создает имитированное хранилище, содержащее объекты `Product`, которые относятся к диапазону категорий. С использованием метода действия `List()` запрашивается одна специфическая категория, а 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 данной 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`

```
...
public void Configure(IApplicationBuilder app,
      IHostingEnvironment env, ILoggerFactory loggerFactory) {
  app.UseDeveloperExceptionPage();
  app.UseStatusCodePages();
  app.UseStaticFiles();
  app.UseMvc(routes => {
    routes.MapRoute(
      name: null,
      template: {category}/Page{page:int},
      defaults: new { controller = Product, action = List }
    );

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

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

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

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

В табл. 9.1 описана схема URL, которую представляют эти 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 дескрипторном вспомогательном классе, созданном в предыдущей главе. Теперь, когда нужно генерировать более сложные URL, необходим способ получения дополнительной информации от представления, не добавляя дополнительные свойства к дескрипторному вспомогательному классу. К счастью, дескрипторные вспомогательные классы обладают удобной возможностью, которая 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`

```
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[page] = 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 дескрипторного вспомогательного класса посредством атрибута `HtmlAttributeName` позволяет указывать префикс для имен атрибутов элемента, которым в данном случае будет `page-url-`. Значение любого атрибута, имя которого начинается с такого префикса, будет добавлено в словарь, присваиваемый свойству `PageUrlValues`. Затем это свойство передается методу `IUrlHelper.Action()` с целью генерации URL для атрибута `href` элементов `a`, которые производит дескрипторный вспомогательный класс.

В листинге 9.5 к элементу `div` добавлен новый атрибут, который обрабатывается дескрипторным вспомогательным классом 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`

```
@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-default
     page-class-selected=btn-primary page-url-category=@Model.CurrentCategory
     class=btn-group pull-right>
</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 необходимо предложить пользователям способ выбора категории, который не предусматривает ввод URL. Это означает, что мы должны отобразить список доступных категорий с отмеченной текущей категорией, если она есть. По мере построения приложения список категорий будет задействован в более чем одном контроллере, поэтому 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 предоставляет 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. Такой подход дает нам обычный класс C#, который может содержать любую необходимую прикладную логику и который можно подвергать модульному тестированию подобно любому другому классу. Это удобный способ создания небольших сегментов приложения, сохраняя общий подход MVC.

Создание навигационного компонента представления

Создайте папку по имени `Components`, которая по соглашению является местом хранения компонентов представлений, и добавьте 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 отде-

льном 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`

```
<!DOCTYPE html>
<html>
<head>
  <meta name=viewport content=width=device-width />
  <link rel=stylesheet asp-href-include=lib/bootstrap/dist/css/*.min.
css />
  <title>SportsStore</title>
</head>
<body>
  <div class=navbar navbar-inverse role=navigation>
    <a class=navbar-brand href=#>SPORTS STORE</a>
  </div>
  <div class=row panel>
    <div id=categories class=col-xs-3>
      @await Component.InvokeAsync(NavigationMenu)
    </div>
    <div class=col-xs-8>
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

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

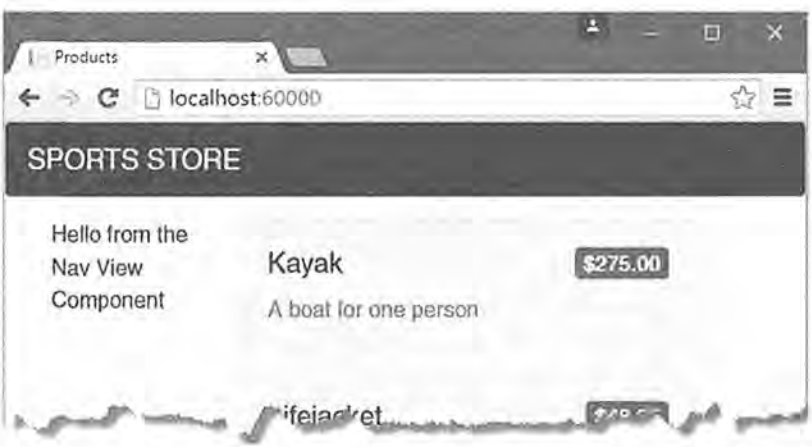

Рис. 9.2. Применение компонента представления

Генерация списков категорий

Теперь можно возвратиться к навигационному компоненту представления и сгенерировать реальный набор категорий. Построить HTML-разметку для категорий можно было бы вручную, как делалось в дескрипторном вспомогательном классе 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 списка категорий в файле NavigationMenuViewComponent.cs

```
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 — предоставление компоненту представления доступа к данным без необходимости знать то, какая реализация хранилища будет использоваться, 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 этого частичного представления возвращаются из метода с применением объекта реализации `IViewComponentResult` (данный процесс будет подробно рассмотрен в главе 22).

Модульное тестирование: генерация списка категорий

Модульный тест, предназначенный для проверки возможности генерации списка категорий, относительно прост. Цель заключается в создании списка, 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 дескрипторному вспомогательному классу 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},
      });

      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

Как будет объясняться в главе 22, 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. Создайте папку `Views/Shared/Components/NavigationMenu` и добавьте 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-default
   asp-action=List
   asp-controller=Product
   asp-route-category=>
     Home
</a>
@foreach (string category in Model) {
  <a class=btn btn-block btn-default
     asp-action=List
     asp-controller=Product
     asp-route-category=@category
     asp-route-page=1>
       @category
  </a>
}
```

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

Запустив приложение, вы увидите 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

```
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},
  });
  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);
}
...
```

Этот модульный тест снабжает компонент представления данными маршрутизации через свойство `ViewComponentContext`, посредством которого компоненты представлений получают все свои данные контекста. Свойство `ViewComponentContext` предоставляет доступ к данным контекста, специфичным для представления, с помощью своего свойства `ViewContext`, которое, в свою очередь, обеспечивает доступ к информации о маршрутизации через свое 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`

```
@model IEnumerable<string>

<a class=btn btn-block btn-default
   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-default)
     asp-action=List
     asp-controller=Product
     asp-route-category=@category
     asp-route-page=1>
       @category
  </a>
}
```

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

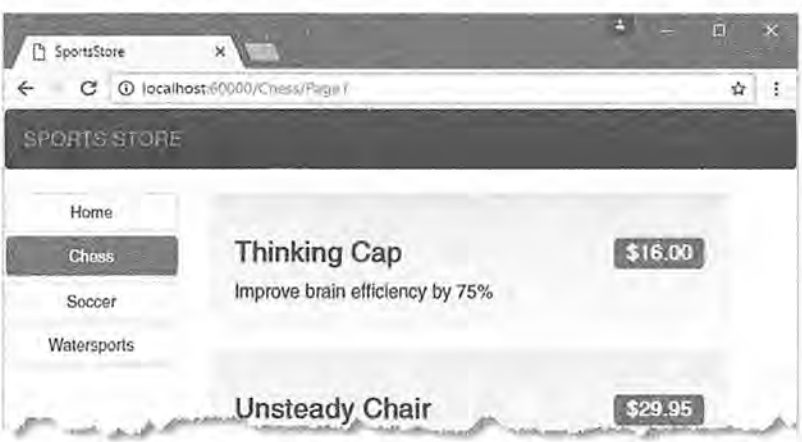

Рис. 9.4. Подсвечивание выбранной категории

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 категории принимались во внимание при разбиении на страницы (листинг 9.12).

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

```
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 page = 1)
      => View(new ProductsListViewModel {
        Products = repository.Products
          .Where(p => category == null || p.Category == category)
          .OrderBy(p => p.ProductID)
          .Skip((page - 1) * PageSize)
          .Take(PageSize),
        PagingInfo = new PagingInfo {
          CurrentPage = page,
          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).

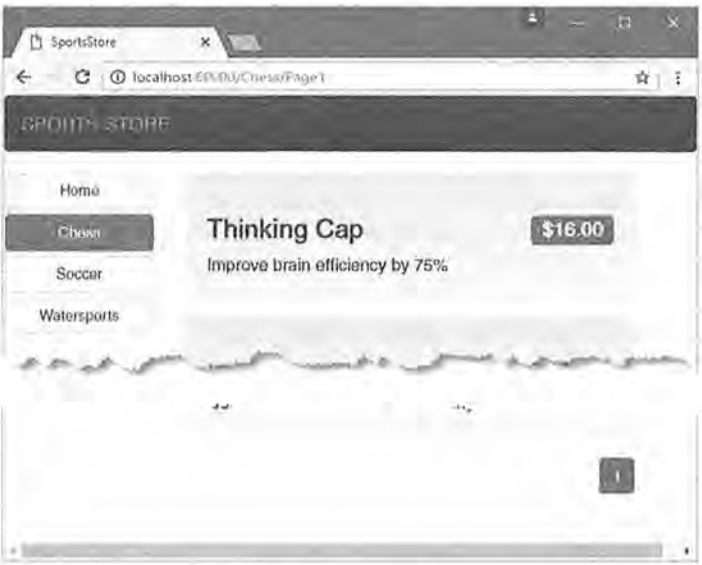

Рис. 9.6. Отображение ссылок на страницы с учетом выбранной категории

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}
  });

  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 продолжает расширяться, но из-за того, что корзина для покупок пока не реализована, продавать товары невозможно. 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.

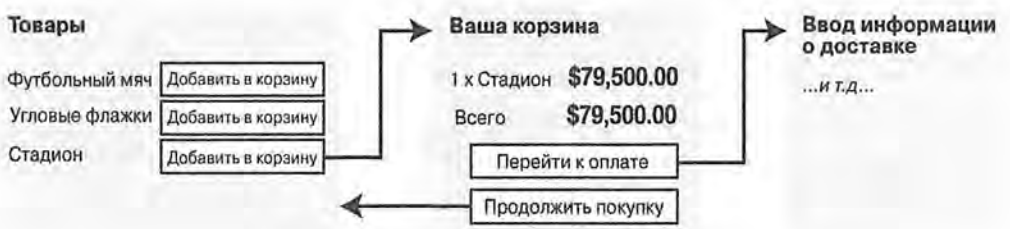

Рис. 9.7. Базовый поток корзины для покупок

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 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

Начните 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 его количество. 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 это легко реализуется 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` создается 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);
}
...
```

Нам также необходимо 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);
}
...
```

Последний тест очень прост. 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. Чтобы подготовиться к этому, добавьте 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()` оперирует над классом `HttpRequest`, используемый инфраструктурой ASP.NET 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 к файлу 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`

```
@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, описывающее каждый товар, которое обновлено для включения кнопки Add To Cart (Добавить в корзину).

Листинг 9.16. Добавление кнопки в файле `ProductSummary.cshtml`

```
@model Product
<div class=well>
  <h3>
    <strong>@Model.Name</strong>
    <span class=pull-right label label-primary>
      @Model.Price.ToString(c)
    </span>
  </h3>
  <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=lead>
      @Model.Description
      <button type=submit class=btn btn-success btn-sm pull-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 дескрипторных вспомогательных классов, 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 эта 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.

Прежде всего, к приложению SportsStore понадобится добавить несколько новых пакетов NuGet. В листинге 9.17 демонстрируются добавления в файле `project.json`.

Листинг 9.17. Добавление пакетов в файле `project.json` внутри проекта `SportsStore`

```
...
dependencies: {
  Microsoft.NETCore.App: {
    version: 1.0.0,
    type: platform
  },
  Microsoft.AspNetCore.Diagnostics: 1.0.0,
  Microsoft.AspNetCore.Server.IISIntegration: 1.0.0,
  Microsoft.AspNetCore.Server.Kestrel: 1.0.0,
  Microsoft.Extensions.Logging.Console: 1.0.0,
  Microsoft.AspNetCore.Razor.Tools: {
    version: 1.0.0-preview2-final,
    type: build
  },
  Microsoft.AspNetCore.StaticFiles: 1.0.0,
  Microsoft.AspNetCore.Mvc: 1.0.0,
  Microsoft.EntityFrameworkCore.SqlServer: 1.0.0,
  Microsoft.EntityFrameworkCore.Tools: 1.0.0-preview2-final,
  Microsoft.Extensions.Configuration.Json: 1.0.0,
  Microsoft.AspNetCore.Session: 1.0.0,
  Microsoft.Extensions.Caching.Memory: 1.0.0,
  Microsoft.AspNetCore.Http.Extensions: 1.0.0
},
...
```

Включение поддержки сеансов требует добавления в класс `Startup` служб и промежуточного программного обеспечения (листинг 9.18).

Листинг 9.18. Включение поддержки сеансов в файле `Startup.cs`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

  public class Startup {
    IConfigurationRoot Configuration;

    public Startup(IHostingEnvironment env) {
      Configuration = new ConfigurationBuilder()
```

```
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile(appsettings.json).Build();
    }
    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, ILoggerFactory loggerFactory) {
      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. Добавьте в папку `Controllers` файл класса по имени `CartController.cs` с определением, представленным в листинге 9.19.

Листинг 9.19. 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 этого контроллера 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. Это идеально подходит для класса `Cart`: мы хотим, чтобы каждый пользователь имел собственную корзину, и эта корзина 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. Добавьте в папку `Infrastructure` файл класса по имени `SessionExtensions.cs` с определениями расширяющих методов, показанными в листинге 9.20.

Листинг 9.20. Содержимое файла `SessionExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Features;
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) эти методы полагаются на пакет Json.NET (глава 20). Пакет Json.NET не потребуется добавлять в файл `package.json`, т.к. он уже используется "за кулисами" инфраструктурой MVC для поддержки средства заголовков JSON, которое описано в главе 21. (Информация о работе с пакетом Json.NET напрямую доступна по адресу `www.newtonsoft.com/json`.)

Расширяющие методы делают сохранение и извлечение объектов `Cart` очень легким. 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 указы-

ваются 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.

Чтобы извлечь объект Cart, применяется другой расширяющий метод, которому указывается тот же самый ключ:

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

Параметр типа позволяет задать 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. Если вы еще раз взглянете на рис. 9.7, то увидите, что это та часть рабочего потока, которая инициируется щелчком 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 этой цели мы создадим простой класс модели представления. Добавьте в папку Models/ViewModels проекта SportsStore файл класса по имени CartIndexViewModel.cs с содержимым, приведенным в листинге 9.21.

Листинг 9.21. 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, как показано в листинге 9.22.

Листинг 9.22. Реализация метода действия Index() в файле CartController.cs

```
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...
  }
}
```

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

Последний шаг при отображении содержимого корзины предусматривает создание представления, которое будет визуализировать действие `Index`. Создайте папку `Views/Cart` и поместите в нее файл представления Razor по имени `Index.cshtml` с разметкой, приведенной в листинге 9.23.

Листинг 9.23. 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 доступна базовая функциональность корзины для покупок. Во-первых, товары выводятся вместе с кнопками Add To Cart (Добавить в корзину), как показано на рис. 9.8.

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) возвратит на страницу товара, из которой произошел переход.

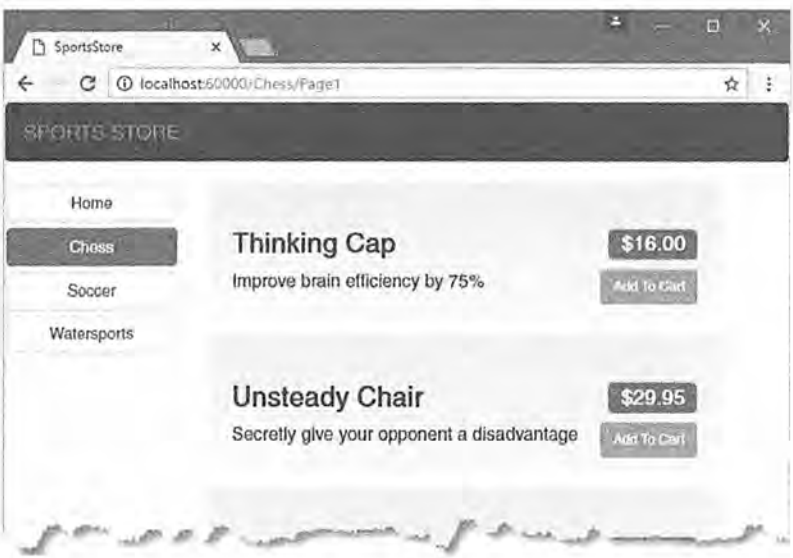

Рис. 9.8. Кнопки Add To Cart

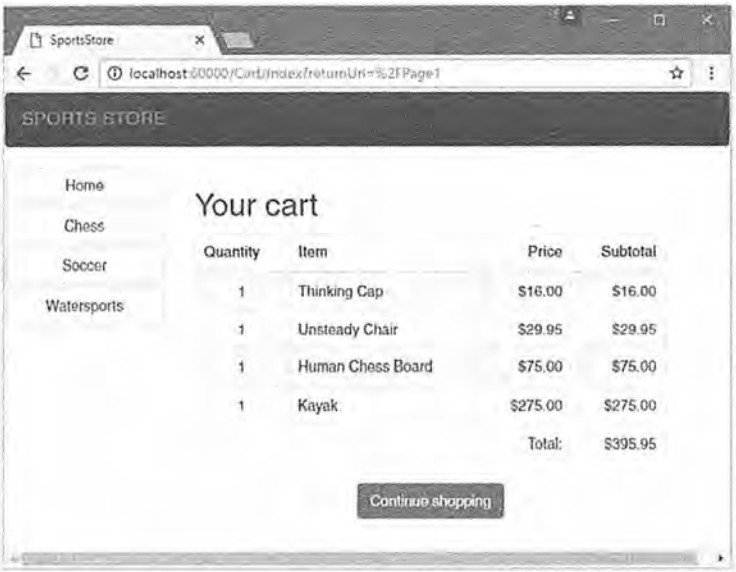

Рис. 9.9. Отображение содержимого корзины для покупок

Резюме

В настоящей 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.