



ASP.NET Core MVC 2

с примерами на C# для профессионалов

*Разрабатывайте оптимизированные под облако веб-приложения
с использованием новейшей версии ASP.NET Core MVC 2 производства Microsoft!*

7-Е ИЗДАНИЕ

Адам Фримен



ДИАЛЕКТИКА

www.williamspublishing.com

apress®

SportsStore: реальное приложение

В предшествующих главах мы создавали очень простое приложение MVC. Был описан паттерн MVC, основные средства языка C#, а также инструменты, необходимые профессиональным разработчикам приложений MVC. Наступило время собрать все вместе и построить несложное, но реалистичное приложение электронной коммерции.

Наше приложение под названием SportsStore будет следовать классическому подходу, который повсеместно используется в онлайн-магазинах. Мы создадим онлайн-каталог товаров, который потребители смогут просматривать по категориям и страницам, корзину для покупок, куда пользователи смогут добавлять и удалять товары, и форму оплаты, где потребители смогут вводить сведения, связанные с доставкой. Кроме того, мы создадим административную область, которая включает в себя средства создания, чтения, обновления и удаления (create, read, update, delete — CRUD) для управления каталогом товаров, и защитим ее так, чтобы изменения могли вносить только зарегистрированные администраторы.

Цель настоящей и последующих глав — дать вам возможность увидеть, на что похожа реальная разработка приложений MVC, за счет создания примера приложения, который максимально приближен к реальности. Разумеется, мы будем ориентироваться на ASP.NET Core MVC, поэтому интеграция с внешними системами, такими как база данных, предельно упрощена, а определенные части приложения, например, обработка платежей, вообще отброшены.

Построение всех уровней необходимой инфраструктуры может показаться несколько медленным, но первоначальные трудозатраты при разработке приложения MVC окупаются, обеспечивая удобный в сопровождении, расширяемый и хорошо структурированный код с великолепной поддержкой модульного тестирования.

Модульное тестирование

Я уже достаточно много говорил о легкости проведения модульного тестирования в MVC, а также о том, что модульное тестирование может быть важной и полезной частью процесса разработки. Это будет демонстрироваться на протяжении данной части книги, поскольку я описываю нюансы и приемы модульного тестирования в их взаимосвязи с основными средствами MVC.

Я понимаю, что мое мнение не является единственно верным. Если вы не хотите прибегать к модульному тестированию, то меня это вполне устроит. Таким образом, когда что-то относится исключительно к тестированию, оно будет помещаться во врезку, подобную настоящей. Если модульное тестирование вас не интересует, тогда можете смело пропускать такие врезки, и приложение SportsStore будет работать не менее успешно. Чтобы воспользоваться преимуществами технологии ASP.NET Core MVC, вовсе не обязательно проводить какое-либо модульное тестирование, хотя поддержка тестирования, конечно же, является основной причиной перехода на ASP.NET Core MVC.

Большинству средств MVC, используемых в приложении SportsStore, посвящены отдельные главы далее в книге. Вместо того чтобы повторять весь материал, я сообщаю ровно столько, сколько требуется для понимания этого примера приложения, и указываю главу, в которой можно почерпнуть более подробные сведения.

Каждый шаг, необходимый для построения приложения, будет выделен, что позволит понять, каким образом средства MVC сочетаются друг с другом. Вы должны уделить особое внимание созданию представлений. Если вы не будете в точности следовать примерам, то получите несколько странные результаты.

Начало работы

Если вы планируете писать код приложения SportsStore на своем компьютере во время изучения материала текущей части книги, тогда вам придется установить Visual Studio и удостовериться в том, что установлен вариант LocalDB, который требуется для постоянного хранения данных. Вариант LocalDB устанавливается автоматически в случае следования инструкциям из главы 2.

На заметку! Если вы просто хотите работать с проектом, не воссоздавая его, то можете загрузить готовый проект SportsStore из хранилища GitHub для настоящей книги (<https://github.com/apress/pro-asp.net-core-mvc-2>). Разумеется, вы вовсе не обязаны повторять все действия. Я старался делать снимки экрана и листинги кода максимально простыми в отслеживании на тот случай, если вы читаете книгу в поезде, кафе или где-то еще.

Создание проекта MVC

Мы будем следовать тому же самому базовому подходу, который использовался в предшествующих главах и заключается в том, чтобы начать с пустого проекта и добавлять в него все необходимые конфигурационные файлы и компоненты. Выберем в меню File (Файл) среды Visual Studio пункт New⇒Project (Создать⇒Проект) и укажем шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), как показано на рис. 8.1. Установим имя проекта в SportsStore и щелкнем на кнопке ОК.

Выберем шаблон Empty (Пустой), как проиллюстрировано на рис. 8.2. Прежде чем щелкать на кнопке ОК для создания проекта SportsStore, удостоверимся в том, что в списках в верхней части окна выбраны варианты .NET Core и ASP.NET Core 2.0, а флажок Enable Docker Support (Включить поддержку Docker) не отмечен.

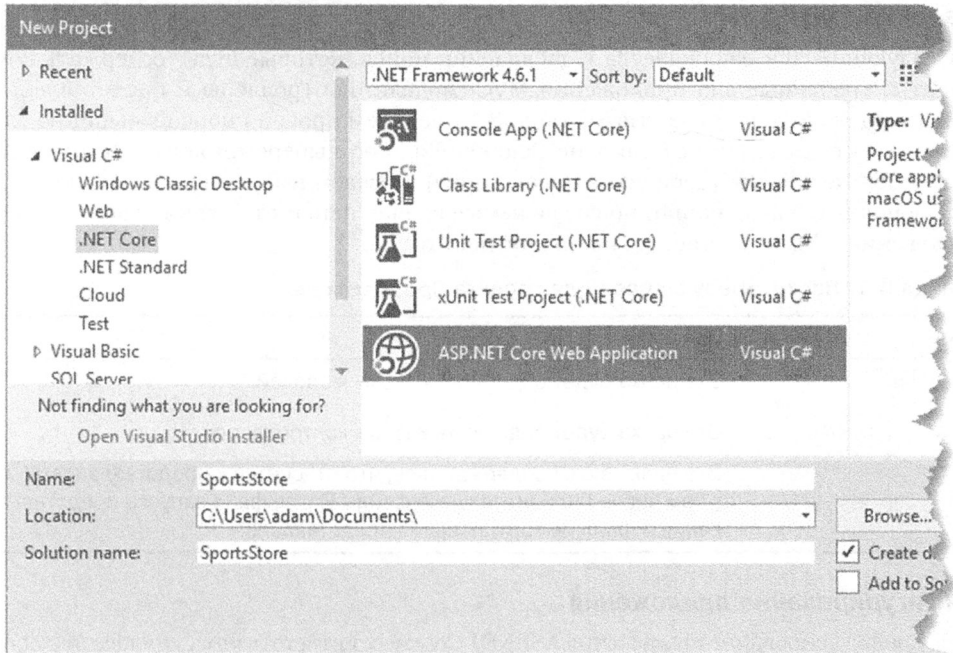


Рис. 8.1. Выбор типа проекта

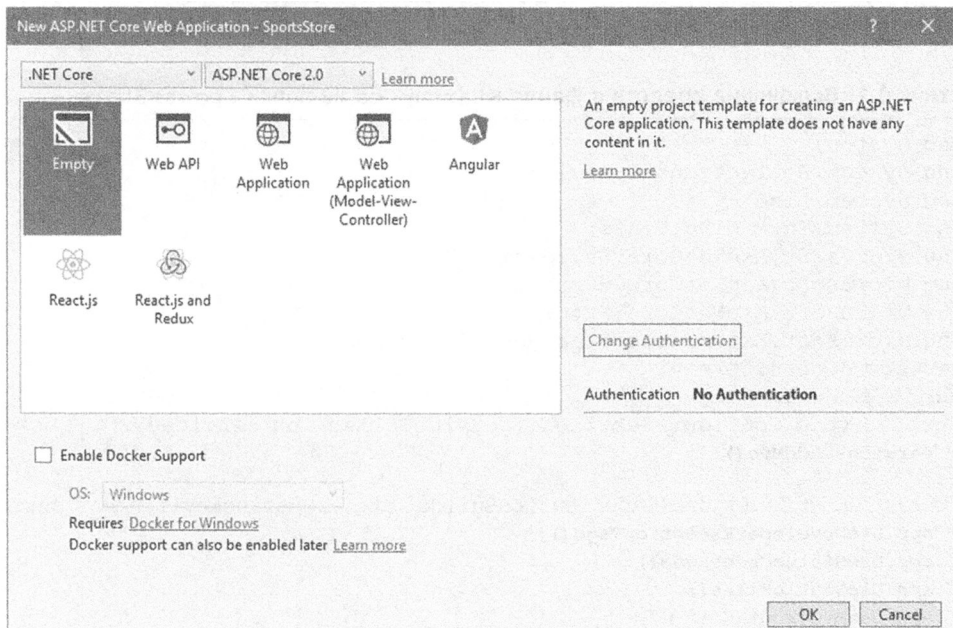


Рис. 8.2. Выбор шаблона проекта

Создание структуры папок

Следующий шаг заключается в добавлении папок, которые будут содержать компоненты, требуемые для приложения MVC: модели, контроллеры и представления. Чтобы создать папки, описанные в табл. 8.1, щелкнем правой кнопкой мыши на элементе проекта SportsStore в окне Solution Explorer, выберем в контекстном меню пункт Add⇒New Folder (Добавить⇒Новая папка) и введем имя папки. Позже потребуются дополнительные папки, но создаваемые сейчас папки отражают главные части приложения MVC и для начала их вполне достаточно.

Таблица 8.1. Папки, требующиеся для проекта SportsStore

Имя	Описание
Models	Эта папка будет содержать классы моделей
Controlllers	Эта папка будет содержать классы контроллеров
Views	Эта папка будет содержать все, что относится к представлениям, в том числе индивидуальные файлы Razor, файл запуска представления и файл импортирования представлений

Конфигурирование приложения

За конфигурирование приложения ASP.NET Core несет ответственность класс Startup. В листинге 8.1 приведены изменения, внесенные в класс Startup для включения инфраструктуры MVC и нескольких связанных средств, которые полезны при разработке.

На заметку! Класс Startup является важным функциональным средством ASP.NET Core и будет подробно описан в главе 14.

Листинг 8.1. Включение средств 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;
namespace SportsStore {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
            });
        }
    }
}
```

Метод `ConfigureServices()` применяется для настройки разделяемых объектов, которые могут использоваться повсеместно в приложении через средство внедрения зависимостей, которое рассматривается в главе 18. Метод `AddMvc()`, вызываемый внутри `ConfigureServices()`, является расширяющим методом, который настраивает разделяемые объекты, применяемые в приложении MVC.

Метод `Configure()` используется для настройки средств, которые получают и обрабатывают HTTP-запросы. Каждый метод, вызываемый в методе `Configure()`, представляет собой расширяющий метод, который настраивает средство обработки HTTP-запросов (табл. 8.2).

Таблица 8.2. Методы настройки первоначальных средств, вызываемые в классе `Startup`

Метод	Описание
<code>UseDeveloperExceptionPage()</code>	Этот расширяющий метод отображает детали исключения, которое произошло в приложении, что полезно во время процесса разработки. Он не должен быть включен в развернутых приложениях и при развертывании приложения в главе 12 будет показано, как отключить данное средство
<code>UseStatusCodePages()</code>	Этот расширяющий метод добавляет простое сообщение в HTTP-ответы, которые иначе не имели бы тела, такие как ответы 404 – Not Found (404 — не найдено)
<code>UseStaticFiles()</code>	Этот расширяющий метод включает поддержку для обслуживания статического содержимого из папки <code>wwwroot</code>
<code>UseMvc()</code>	Этот расширяющий метод включает инфраструктуру ASP.NET Core MVC

Далее понадобится подготовить приложение для представлений Razor. Щелкнем правой кнопкой мыши на папке `Views`, выберем в контекстном меню пункт `Add⇒New Item` (Добавить⇒Новый элемент) и укажем шаблон `MVC View Imports Page` (Страница импортирования представлений MVC) из категории `ASP.NET` (рис. 8.3).

Щелкнем на кнопке `Add` (Добавить), чтобы создать файл `_ViewImports.cshtml`, и приведем его содержимое в соответствие с листингом 8.2.

Листинг 8.2. Содержимое файла `_ViewImports.cshtml` из папки `Views`

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

Оператор `@using` позволит применять типы из пространства имен `SportsStore.Models` в представлениях, не ссылаясь на это пространство имен. Оператор `@addTagHelper` включает встроенные вспомогательные функции дескрипторов, которые будут использоваться позже для создания элементов HTML, отражающих конфигурацию приложения `SportsStore`.

Создание проекта модульного тестирования

Создание проекта модульного тестирования требует выполнения того же самого процесса, который был описан в главе 7. Щелкнем правой кнопкой мыши на элементе решения `SportsStore` в окне `Solution Explorer` и выберем в контекстном меню пункт `Add⇒New Project` (Добавить⇒Новый проект).

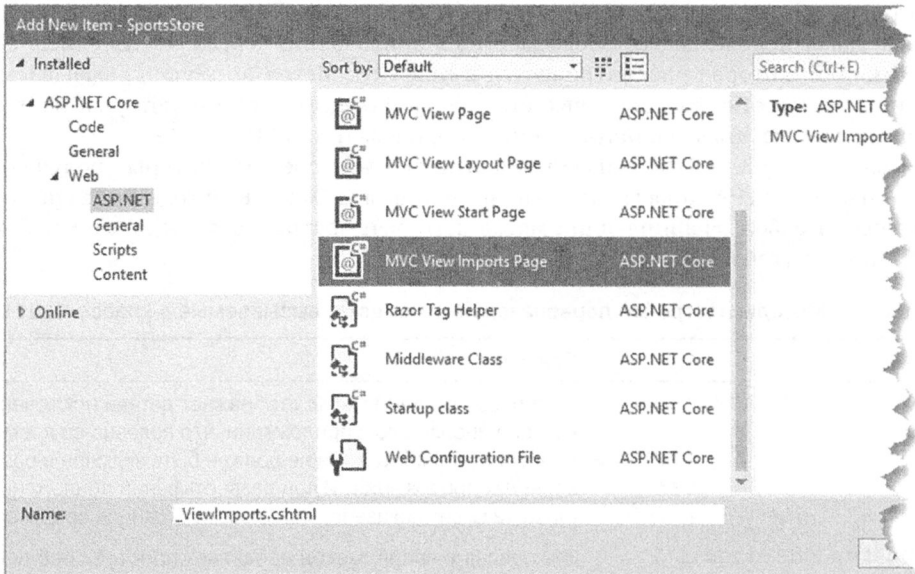


Рис. 8.3. Создание файла импортирования представлений

Укажем шаблон **xUnit Test Project (.NET Core)** (Проект тестирования xUnit (.NET Core)), как показано на рис. 8.4, и установим имя проекта в `SportsStore.Tests`. Щелкнем на кнопке **OK**, чтобы создать проект модульного тестирования.

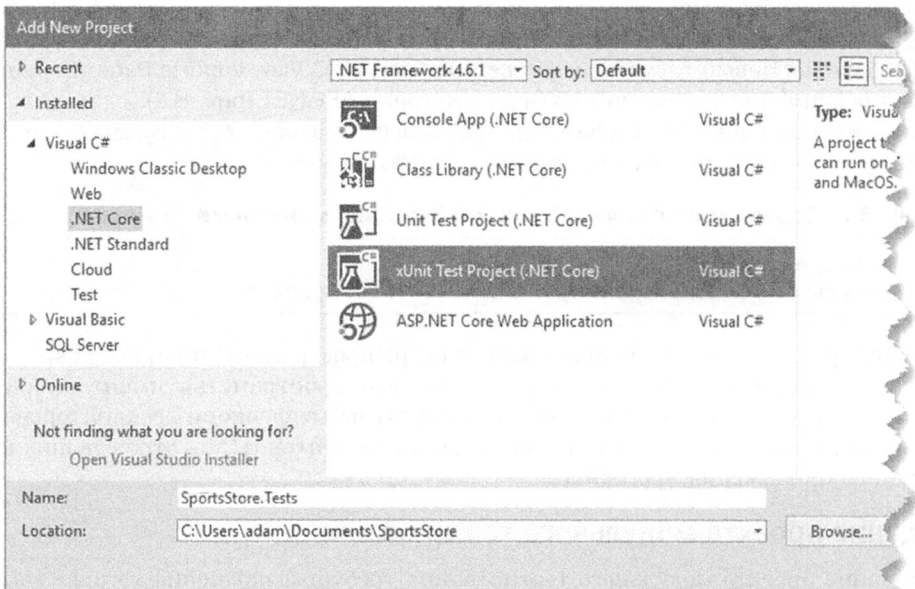


Рис. 8.4. Создание проекта модульного тестирования

После того, как проект модульного тестирования создан, щелчком правой кнопкой мыши на проекте SportsStore.Tests в окне Solution Explorer и выберем в контекстном меню пункт Edit SportsStore.Tests.csproj (Редактировать SportsStore.Tests.csproj). Внесем в файл SportsStore.Tests.csproj изменения, показанные в листинге 8.3, чтобы добавить пакет Moq в проект SportsStore.Tests и создать ссылку на главный проект SportsStore. Удостоверимся, что для пакета Moq указана та же версия, что и в листинге 8.3.

Листинг 8.3. Добавление пакета в файле SportsStore.Tests.csproj из папки SportsStore.Tests

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\SportsStore\SportsStore.csproj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
      Version="15.3.0-preview-20170628-02" />
    <PackageReference Include="xunit" Version="2.2.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
    <PackageReference Include="Moq" Version="4.7.99" />
  </ItemGroup>
</Project>
```

Как только изменения будут сохранены, среда Visual Studio загрузит и установит пакет Moq в проект модульного тестирования и создаст ссылку на главный проект SportsStore, так что содержащиеся в нем классы смогут использоваться в тестах.

Проверка и запуск приложения

Проекты приложения и модульного тестирования созданы, сконфигурированы и готовы к дальнейшей разработке. Окно Solution Explorer должно содержать элементы, представленные на рис. 8.5. Если вы видите другие элементы или элементы расположены не в тех позициях, тогда возникнут проблемы, поэтому уделите время проверке, что все элементы присутствуют и находятся на своих местах.

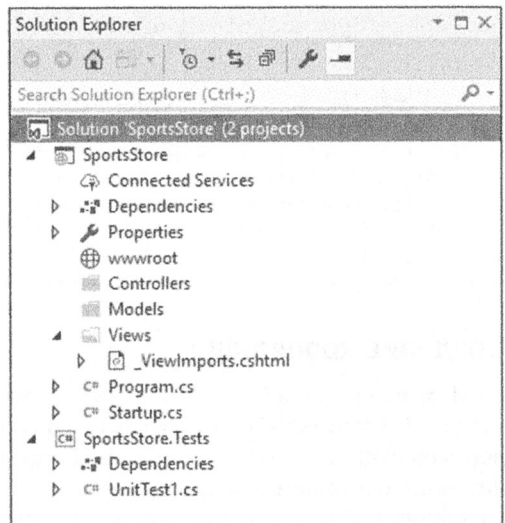


Рис. 8.5. Окно Solution Explorer для проектов приложения SportsStore и модульного тестирования

В результате выбора в меню Debug (Отладка) пункта Start Debugging (Запустить отладку) или Start Without Debugging (Запустить без отладки), если предпочтение отдается итеративному стилю разработки, описанному в главе 6, появится страница ошибки (рис. 8.6). Сообщение об ошибке отображается из-за того, что в настоящий момент в приложении нет контроллеров для обработки запросов; проблему мы вскоре устраним.

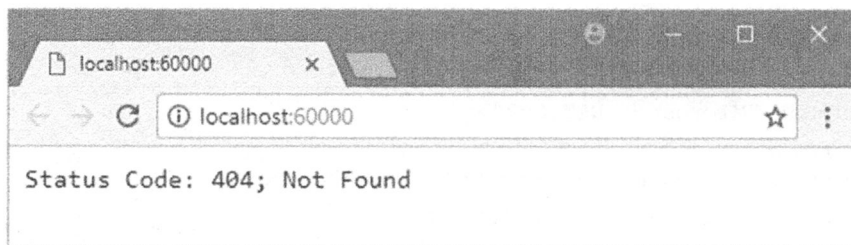


Рис. 8.6. Выполнение приложения SportsStore

Начало работы с моделью предметной области

Все проекты начинаются с создания модели предметной области, которая является центральной частью приложения MVC. Поскольку мы строим приложение электронной коммерции, то наиболее очевидная модель, которая необходима, относится к товару. Добавим в папку Models файл класса по имени Product.cs с определением, приведенным в листинге 8.4.

Листинг 8.4. Содержимое файла Product.cs из папки Models

```
namespace SportsStore.Models {
    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

Создание хранилища

Нам нужен какой-нибудь способ получения объектов Product из базы данных. В главе 3 упоминалось, что модель включает в себя логику для сохранения и извлечения данных из постоянного хранилища. Пока можно не беспокоиться о том, как будет реализовано постоянство данных, но необходимо начать процесс определения интерфейса для него. Добавим в папку Models новый файл по имени IProductRepository.cs и поместим в него определение интерфейса, показанное в листинге 8.5.

Листинг 8.5. Содержимое файла `IProductRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {
    public interface IProductRepository {
        IQueryable<Product> Products { get; }
    }
}
```

Интерфейс `IProductRepository` использует `IQueryable<T>`, чтобы позволить вызывающему коду получать последовательность объектов `Product`. Интерфейс `IQueryable<T>` является производным от более знакомого интерфейса `IEnumerable<T>` и представляет коллекцию объектов, которые можно запрашивать, вроде тех, что управляются базой данных.

Класс, зависящий от интерфейса `IProductRepository`, способен получать объекты `Product` без необходимости знать детали того, как они хранятся, или каким образом класс реализации будет их доставлять.

Интерфейсы `IEnumerable<T>` и `IQueryable<T>`

Интерфейс `IQueryable<T>` удобен, потому что он позволяет эффективно организовать запрашивание коллекции объектов. Позже в главе будет добавлена поддержка для извлечения из базы данных подмножества объектов `Product`. Применение интерфейса `IQueryable<T>` дает возможность запрашивать у базы данных именно те объекты, которые требуются, с помощью стандартных операторов LINQ, не нуждаясь в информации о том, какой сервер баз данных хранит данные или как он обрабатывает запрос. Без интерфейса `IQueryable<T>` пришлось бы извлекать из базы данных все объекты `Product` и затем отбрасывать ненужные, что с ростом объема данных, используемых приложением, превращается в затратную операцию. Именно по этой причине в интерфейсах и классах хранилища в форме базы данных обычно вместо `IEnumerable<T>` применяется `IQueryable<T>`.

Однако во время использования интерфейса `IQueryable<T>` следует соблюдать осторожность, поскольку каждый раз, когда происходит перечисление коллекции объектов, запрос будет оцениваться заново, т.е. базе данных отправится новый запрос. В результате выгоды в плане эффективности от применения `IQueryable<T>` могут быть сведены на нет. В таких ситуациях `IQueryable<T>` можно преобразовать в более предсказуемую форму, используя расширяющий метод `ToList()` или `ToArray()`.

Создание фиктивного хранилища

Теперь, когда определен интерфейс, можно было бы реализовать механизм постоянного и привязать его к базе данных, но сначала необходимо добавить ряд других частей приложения. Для этого мы создадим фиктивную реализацию интерфейса `IProductRepository`, которая будет замещать хранилище данных до тех пор, пока мы им не займемся. Чтобы создать фиктивное хранилище, добавим в папку `Models` файл класса по имени `FakeProductRepository.cs` и поместим в него определение, представленное в листинге 8.6.

Листинг 8.6. Содержимое файла 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>();
    }
}
```

Класс `FakeProductRepository` реализует интерфейс `IProductRepository`, возвращая в качестве значения свойства `Products` фиксированную коллекцию объектов `Product`. Метод `AsQueryable()` применяется для преобразования фиксированной коллекции объектов в `IQueryable<Product>`, чтобы реализовать интерфейс `IProductRepository` и позволить создавать совместимое фиктивное хранилище, не имея дело с реальными запросами.

Регистрация службы хранилища

В MVC придается особое значение использованию *слабо связанных компонентов*, что означает возможность вносить изменения в одну часть приложения, не внося соответствующие изменения где-то в другом месте. Такой подход трактует части приложения как *службы*, предлагающие функциональные средства, которые задействуются в других частях приложения. Класс, предоставляющий службу, впоследствии может быть модифицирован или заменен, не требуя внесения изменений в классы, в которых он применяется. Подход более подробно объясняется в главе 18, а в случае приложения `SportsStore` понадобится создать службу хранилища, которая позволит контроллерам получать реализующие интерфейс `IProductRepository` объекты, не зная, какой класс используется. В итоге появится возможность начать разработку приложения с применением простого класса `FakeProductRepository`, созданного в предыдущем разделе, и позже заменить его реальным хранилищем, не внося изменения во все классы, которым необходим доступ в хранилище. Службы регистрируются в методе `ConfigureServices()` класса `Startup`; в листинге 8.7 определена новая служба для хранилища.

Листинг 8.7. Создание службы хранилища 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;

namespace SportsStore {
```

```

public class Startup {
    public void ConfigureServices(IServiceCollection services) {
        services.AddTransient<IProductRepository, FakeProductRepository>();
        services.AddMvc();
    }

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

```

Добавленный в метод `ConfigureServices()` оператор сообщает инфраструктуре ASP.NET Core о том, что когда компоненту вроде контроллера необходима реализация интерфейса `IProductRepository`, она должна получить экземпляр класса `FakeProductRepository`. Метод `AddTransient()` указывает, что каждый раз, когда требуется реализация интерфейса `IProductRepository`, должен создаваться новый объект `FakeProductRepository`. Не беспокойтесь, если смысл кода пока не особенно понятен: вскоре вы увидите, как он вписывается в приложение, а детали того, что происходит, будут представлены в главе 18.

Отображение списка товаров

Оставшийся материал главы можно было бы посвятить построению модели предметной области и хранилища, вообще не касаясь других частей приложения. Однако такой подход выглядит довольно скучным, поэтому мы пойдем другим путем и приступим к интенсивному использованию MVC, вернувшись к добавлению функциональности модели и хранилища, когда она понадобится.

В текущем разделе нам предстоит создать контроллер и метод действия, который может отображать сведения о товарах в хранилище. Пока ими будут только данные из фиктивного хранилища, но позже мы решим эту задачу. Мы также подготовим начальную *конфигурацию маршрутизации*, чтобы инфраструктура MVC знала, каким образом сопоставлять запросы в приложении с контроллером, который будет создан.

Использование инструмента формирования шаблонов MVC в Visual Studio

Везде в книге контроллеры и представления MVC создаются за счет щелчка правой кнопкой мыши на папке в окне Solution Explorer, выбора в контекстном меню пункта `Add⇒New Item` (Добавить⇒Новый элемент) и указания шаблона элемента в открывшемся диалоговом окне `Add New Item` (Добавление нового элемента). Существует альтернативный прием, называемый *формированием шаблонов* (scaffolding), при котором среда Visual Studio предлагает в контекстном меню `Add` (Добавить) пункты, специально предназначенные для создания контроллеров и представлений. После выбора таких пунктов меню предлагается выбрать сценарий для создаваемого компонента, такой как контроллер с действиями, допускающими чтение/запись, или представление, которое содержит форму, применяемую для создания специфического объекта модели.

Формирование шаблонов в книге не используется. Код и разметка, генерируемые инструментом формирования шаблонов, являются настолько общими, что оказываются едва ли не бесполезными, в то время как набор поддерживаемых сценариев ограничен и не решает распространенные задачи разработки. Цель настоящей книги — не только донести до вас знания, каким образом создавать приложения MVC, но также объяснить, как все работает “за кулисами”, и сделать это гораздо труднее, когда ответственность за создание компонентов возлагается на инструмент формирования шаблонов.

Тем не менее, мы имеем еще одну ситуацию, когда ваш стиль разработки может отличаться от моего, и в своих проектах вы вполне можете предпочесть работу с инструментом формирования шаблонов. Ваше решение полностью приемлемо, хотя я рекомендую уделить время на выяснение того, что именно инструмент формирования шаблонов делает, чтобы знать, куда затем смотреть, если будут получены не те результаты, которые ожидались.

Добавление контроллера

Чтобы создать первый контроллер в приложении, добавим в папку `Controllers` файл класса по имени `ProductController.cs` с определением, показанным в листинге 8.8.

Листинг 8.8. Содержимое файла `ProductController.cs` из папки `Controllers`

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

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

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

Когда инфраструктуре MVC необходимо создать новый экземпляр класса `ProductController` для обработки HTTP-запроса, MVC проинспектирует конструктор и выяснит, что он требует объекта, который реализует интерфейс `IProductRepository`. Чтобы определить, какой класс реализации должен использоваться, инфраструктура MVC обращается к конфигурации в классе `Startup`, которая сообщает о том, что нужно применять класс `FakeRepository` и каждый раз создавать его новый экземпляр. Инфраструктура MVC создает новый объект `FakeRepository` и использует его для вызова конструктора `ProductController` с целью создания объекта контроллера, который будет обрабатывать HTTP-запрос.

Такой подход известен под названием *внедрение зависимостей* и позволяет объекту `ProductController` получать доступ к хранилищу приложения через интерфейс `IProductRepository` без необходимости в знании того, какой класс реализации был сконфигурирован. Позже мы заменим фиктивное хранилище реальным, а благодаря внедрению зависимостей контроллер продолжит работать безо всяких изменений.

На заметку! Некоторым разработчикам не нравится внедрение зависимостей, поскольку они считают, что оно усложняет приложения. Я не придерживаюсь такой точки зрения, но если вы не знакомы с внедрением зависимостей, тогда рекомендую вам прочитать главу 18, прежде чем принимать решение.

Далее добавим метод действия по имени `List()`, который будет визуализировать представление, отображающее полный список товаров из хранилища (листинг 8.9).

Листинг 8.9. Добавление метода действия в файле `ProductController.cs` из папки `Controllers`

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

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

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

        public IActionResult List() => View(repository.Products);
    }
}
```

Вызов метода `View()` подобного рода (без указания имени представления) сообщает инфраструктуре MVC о том, что нужно визуализировать стандартное представление для метода действия. Передача методу `View()` коллекции объектов `Product` из хранилища снабжает инфраструктуру данными, которыми необходимо заполнить объект `Model` в строго типизированном представлении.

Добавление и конфигурирование представления

Нам требуется создать представление для отображения содержимого пользователя, но предстоит проделать ряд подготовительных шагов, чтобы упростить написание представления. Сначала понадобится создать разделяемую компоновку, в которой будет определено общее содержимое, включаемое во все отправляемые клиентам HTML-ответы. Разделяемые компоновки — удобный способ обеспечить согласованность представлений и наличие в них важных файлов JavaScript и таблиц стилей CSS; их работа объяснялась в главе 5.

Создадим папку `Views/Shared` и добавим в нее новый файл с шаблоном MVC `View Layout Page` (Страница компоновки представлений MVC) и именем `_Layout.cshtml`, которое является стандартным именем, назначаемым средой Visual Studio элементу такого типа. В листинге 8.10 приведено содержимое файла `_Layout.cshtml`. В стандартное содержимое внесено одно изменение, связанное с установкой внутренностей элемента `title` в `SportsStore`.

Листинг 8.10. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
```

```

<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>

```

Затем необходимо сконфигурировать приложение, чтобы файл `_Layout.cshtml` применялся по умолчанию, для чего в папку `Views` добавляется файл с шаблоном MVC View Start Page (Файл запуска представления MVC) и именем `_ViewStart.cshtml`. Стандартное содержимое, добавляемое Visual Studio (листинг 8.11), выбирает компоновку по имени `_Layout.cshtml`, которая соответствует файлу из листинга 8.10.

Листинг 8.11. Содержимое файла `_ViewStart.cshtml` из папки `Views`

```

@{
  Layout = "_Layout";
}

```

Теперь нужно добавить представление, которое будет отображаться, когда для обработки запроса используется метод действия `List()`. Создадим папку `Views/Product` и добавим в нее файл представления Razor по имени `List.cshtml`. Поместим в него разметку, показанную в листинге 8.12.

Листинг 8.12. Содержимое файла `List.cshtml` из папки `Views/Product`

```

@model IEnumerable<Product>
@foreach (var p in Model) {
  <div>
    <h3>@p.Name</h3>
    @p.Description
    <h4>@p.Price.ToString("c")</h4>
  </div>
}

```

Выражение `@model` в начале файла указывает, что представление будет получать от метода действия последовательность объектов `Product` в качестве данных модели. С помощью выражения `@foreach` осуществляется проход по этой последовательности с генерацией простого набора HTML-элементов для каждого полученного объекта `Product`.

Представление не знает, откуда поступили объекты `Product`, как они были получены или охватывают ли они все товары, известные приложению. Взамен представление имеет дело только со способом отображения деталей каждого объекта `Product` с применением HTML-элементов, что согласуется с принципом разделения обязанностей, который был описан в главе 3.

Совет. Значение свойства `Price` преобразуется в строку с использованием метода `ToString("c")`, который визуализирует числовые значения в виде денежных значений в соответствии с настройками культуры, действующими на сервере. Например, если в качестве настройки культуры сервера установлено `en-US`, то вызов `(1002.3).ToString("c")` возвратит значение `$1,002.30`, но если для сервера установлена культура `en-GB`, тогда тот же самый вызов возвратит `£1,002.30`.

Установка стандартного маршрута

Инфраструктуре MVC понадобится сообщить, что она должна отправлять запросы, поступающие для корневого URL приложения (`http://мой-сайт/`), методу действия `List()` класса `ProductController`. Задача решается редактированием оператора в классе `Startup`, который настраивает классы MVC, обрабатывающие HTTP-запросы (листинг 8.13).

Листинг 8.13. Изменение стандартного маршрута 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;

namespace SportsStore {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IProductRepository, FakeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}
```

Метод `Configure()` класса `Startup` применяется для настройки конвейера запросов, состоящего из классов (известных как *промежуточное программное обеспечение*), которые будут инспектировать HTTP-запросы и генерировать ответы.

Метод `UseMvc()` настраивает промежуточное программное обеспечение MVC, причем одним из параметров конфигурации является схема, которая будет использоваться для сопоставления URL с контроллерами и методами действий. Система маршрутизации подробно рассматривается в главах 15 и 16, а пока просто следует знать, что изменения, выделенные в листинге 8.13, сообщают инфраструктуре MVC о необходимости отправки запросов методу действия `List()` контроллера `Product`, если только в URL запроса не указано иное.

Совет. Обратите внимание, что в листинге 8.13 имя контроллера установлено в `Product`, а не `ProductController`, являющееся именем класса. Это часть соглашения об именовании MVC, в рамках которого имена классов контроллеров обычно заканчиваются словом `Controller`, но при ссылке на класс данная часть имени опускается. Соглашение об именовании и его влияние объясняются в главе 31.

Запуск приложения

Все основные компоненты в наличии. Мы имеем контроллер с методом действия, который MVC будет применять, когда запрашивается стандартный URL для приложения. Инфраструктура MVC создаст экземпляр класса `FakeRepository`, после чего будет использовать его для создания нового объекта контроллера, обрабатывающего запрос. Фиктивное хранилище снабдит контроллер простыми тестовыми данными, которые его метод действия передаст представлению `Razor`, так что HTML-ответ для браузера будет включать детали каждого объекта `Product`. При генерации HTML-ответа инфраструктура MVC объединит данные из представления, выбранные методом действия, с содержимым разделяемой компоновки, порождая заверченный HTML-документ, который браузер в состоянии разобрать и отобразить. Запустив приложение, можно увидеть результат, показанный на рис. 8.7.

Итак, мы исследовали типовой шаблон разработки для инфраструктуры ASP.NET Core MVC. Начальные затраты времени на необходимую настройку являются обязательными, но затем основные средства приложения будут собираться очень быстро.



Рис. 8.7. Просмотр основной функциональности приложения

Подготовка базы данных

Мы способны отображать простое представление, содержащее сведения о товарах, но применяем тестовые данные, которые находятся в фиктивном хранилище. Прежде чем можно будет реализовать хранилище с реальными данными, необходимо настроить базу данных и заполнить ее данными.

Для базы данных будет использоваться SQL Server, а доступ к ней будет осуществляться с применением Entity Framework Core (EF Core) — инфраструктуры объектно-реляционного отображения (object-relational mapping — ORM) для Microsoft .NET. Инфраструктура ORM представляет таблицы, столбцы и строки реляционной базы данных посредством обычных объектов C#.

На заметку! Это область, где можно выбирать из широкого диапазона инструментальных средств и технологий. Доступны не только различные реляционные базы данных, но можно также работать с хранилищами объектов, хранилищами документов и рядом экзотических альтернатив. Кроме того, существуют другие инфраструктуры ORM для .NET, каждая из которых использует слегка отличающийся подход; такие вариации позволяют выбрать то, что лучше всего подходит для конкретных проектов.

Инфраструктура Entity Framework Core применяется по нескольким причинам: ее просто запустить в работу, она прекрасно интегрируется с LINQ (а мне нравится использовать LINQ) и она хорошо работает с ASP.NET Core MVC. Ранним выпускам EF Core были присущи мелкие недостатки, но текущие версии элегантны и богаты возможностями.

Продукт SQL Server располагает удобным средством *LocalDB*, которое представляет собой не требующую администрирования реализацию основной функциональности SQL Server, специально спроектированную для разработчиков. Благодаря LocalDB можно пропускать процесс настройки базы данных на время построения проекта, а развертывание проводить в полном экземпляре SQL Server позже. Большинство приложений MVC развертываются в размещаемых средах, которые обслуживаются профессиональными администраторами, и наличие средства LocalDB означает, что конфигурирование баз данных остается в руках администраторов баз данных, позволяя разработчикам приложений заниматься написанием кода.

Совет. Если при установке Visual Studio вы не выбрали LocalDB, то придется сделать это сейчас. Средство LocalDB может быть выбрано через раздел Individual Components (Индивидуальные компоненты) в программе установки Visual Studio. Если вы следовали инструкциям, приведенным в главе 2, тогда средство LocalDB должно быть установлено и готово к применению.

Установка пакета инструментов командной строки Entity Framework Core

Основная функциональность Entity Framework Core добавляется в проект по умолчанию, когда Visual Studio создает проект. Нам необходим один дополнительный пакет NuGet, предоставляющий инструменты командной строки, которые используются для создания классов, готовящих базу данных к хранению данных приложения. Такие классы известны как *миграции*.

Чтобы добавить пакет в проект, щелчком правой кнопкой мыши на элементе проекта `SportsStore` в окне `Solution Explorer`, выберем в контекстном меню пункт `Edit SportsStore.csproj` (Редактировать `SportsStore.csproj`) и внесем в файл `SportsStore.csproj` изменения, показанные в листинге 8.14. Важно применять версию, указанную в листинге. Кроме того, пакет добавляется с использованием элемента `DotNetCliToolReference`, а не `PackageReference`, который предназначен для добавления существующего пакета.

На заметку! Пакет инструментов командной строки `Entity Framework Core` должен устанавливаться за счет редактирования файла `SportsStore.csproj`. Пакеты такого типа не могут быть добавлены с применением диспетчера пакетов `NuGet` либо инструмента командной строки `dotnet`.

Листинг 8.14. Добавление пакета в файле `SportsStore.csproj` из папки `SportsStore`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="2.0.0" />
  </ItemGroup>
</Project>
```

После сохранения файла среда `Visual Studio` загрузит и установит пакет инструментов командной строки `Entity Framework Core`, а также добавит их в проект.

Создание классов базы данных

Класс контекста базы данных — это шлюз между приложением и `EF Core`, который обеспечивает доступ к данным приложения с использованием объектов моделей. Чтобы создать класс контекста базы данных для приложения `SportsStore`, добавим в папку `Models` файл класса по имени `ApplicationDbContext.cs` с определением, приведенным в листинге 8.15.

Листинг 8.15. Содержимое файла `ApplicationDbContext.cs` из папки `Models`

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

namespace SportsStore.Models {
    public class ApplicationDbContext : DbContext {
```

```

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

public DbSet<Product> Products { get; set; }
}
}

```

Базовый класс `DbContext` предоставляет доступ к лежащей в основе функциональности `Entity Framework Core`, а свойство `Products` обеспечивает доступ к объектам `Product` в базе данных. Класс `ApplicationDbContext` является производным от `DbContext` и добавляет свойства, которые будут применяться для чтения и записи данных приложения. В текущий момент имеется только одно свойство, которое предоставит доступ к объектам.

Создание класса хранилища

Хотя в настоящее время может показаться иначе, но большая часть работы, требуемой для настройки базы данных, завершена. Следующий шаг заключается в создании класса, который реализует интерфейс `IProductRepository` и получает данные с использованием инфраструктуры `Entity Framework Core`. Добавим в папку `Models` файл класса по имени `EFProductRepository.cs` с определением класса хранилища, представленным в листинге 8.16.

Листинг 8.16. Содержимое файла `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;
    }
}

```

По мере добавления средств к приложению в класс будет добавляться соответствующая функциональность, а пока реализация хранилища просто отображает свойство `Products`, определенное в интерфейсе `IProductRepository`, на свойство `Products`, которое определено в классе `ApplicationDbContext`. Свойство `Products` в классе контекста возвращает объект `DbSet<Product>`, который реализует интерфейс `IQueryable<T>` и облегчает реализацию интерфейса `IProductRepository`, когда применяется `Entity Framework Core`. В результате гарантируется, что запросы к базе данных будут извлекать только те объекты, которые требуются, как объяснялось ранее в главе.

Определение строки подключения

Строка подключения указывает местоположение и имя базы данных, а также предоставляет конфигурационные настройки, с которыми приложение должно под-

ключаться к серверу базы данных. Строки подключения хранятся в файле JSON под названием `appsettings.json`, который создан в проекте `SportsStore` с использованием шаблона элемента ASP.NET Configuration File (Конфигурационный файл ASP.NET) из раздела General (Общие) диалогового окна Add New Item.

При создании файла `appsettings.json` среда Visual Studio добавляет в него заполнитель для строки подключения, который необходимо заменить содержимым листинга 8.17.

Совет. Строка подключения должна быть выражена как единственная неразрывная строка, что выглядит нормально в редакторе Visual Studio, но не укладывается на печатной странице, чем и объясняется неуклюжее форматирование в листинге 8.17. При определении строки подключения в собственном проекте удостоверьтесь, что значение элемента `ConnectionString` находится в одной строке.

Листинг 8.17. Редактирование строки подключения в файле `appsettings.json` из папки `SportsStore`

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

Внутри раздела `Data` конфигурационного файла имя строки подключения устанавливается в `SportStoreProducts`. Значение элемента `ConnectionString` указывает, что для базы данных по имени `SportsStore` должно применяться средство `LocalDB`.

Конфигурирование приложения

Далее необходимо прочитать строку подключения и сконфигурировать приложение для ее использования при подключении к базе данных. В листинге 8.18 показаны изменения, которые потребуется внести в класс `Startup`, чтобы получать детали данных конфигурации, содержащихся в файле `appsettings.json`, и применять их для конфигурирования Entity Framework Core. (Работа, связанная с чтением файла JSON, обеспечивается классом `Program`, который будет описан в главе 14.)

Листинг 8.18. Конфигурирование приложения 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();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}

```

Добавленный в класс `Startup` конструктор получает данные конфигурации из файла `appsettings.json`, которые представлены посредством объекта, реализующего интерфейс `IConfiguration`. Конструктор присваивает объект реализации `IConfiguration` свойству по имени `Configuration`, так что он может использоваться в остальном коде класса `Startup`.

В главе 14 будет показано, как читать и обращаться к данным конфигурации. Для приложения `SportsStore` мы добавили в метод `ConfigureServices()` последовательность вызовов методов, которая настраивает `Entity Framework Core`:

```

...
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        Configuration["Data:SportStoreProducts:ConnectionString"]));
...

```

Расширяющий метод `AddDbContext()` настраивает службы, предоставляемые инфраструктурой `Entity Framework Core`, для контекста базы данных, который был создан в листинге 8.15. Как объясняется в главе 14, многие методы, применяемые в классе `Startup`, позволяют конфигурировать службы и средства промежуточного программного обеспечения с использованием аргументов `options`. Аргументом метода `AddDbContext()` является лямбда-выражение, которое получает объект `options`, конфигурирующий базу данных для класса контекста. В этом случае база данных конфигурируется с помощью метода `UseSqlServer()` и указания строки подключения, которая получена из свойства `Configuration`.

Еще одним изменением, внесенным в класс `Startup`, была замена фиктивного хранилища реальным:

```
...
services.AddTransient<IProductRepository, EFProductRepository>();
...
```

Компоненты в приложении, работающие с интерфейсом `IProductRepository`, к которым в настоящий момент относится только контроллер `Product`, при создании будут получать объект `EFProductRepository`, предоставляющий им доступ к информации в базе данных. Подробные объяснения приводятся в главе 18, а пока просто знайте, что результатом будет гладкая замена фиктивных данных реальными из базы данных без необходимости в изменении класса `ProductController`.

Отключение проверки области видимости

Применение Entity Framework Core требует внесения изменения в конфигурацию средства внедрения зависимостей, которое рассматривается в главе 18. Класс `Program` несет ответственность за запуск и конфигурирование ASP.NET Core перед передачей управления классу `Startup`, и в листинге 8.19 показано необходимое изменение. Без такого изменения попытка создать схему базы данных в следующем разделе приведет к генерации исключения.

Листинг 8.19. Подготовка для работы с Entity Framework Core в файле `Program.cs` из папки `SportsStore`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace SportsStore {
    public class Program {
        public static void Main(string[] args) {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .UseDefaultServiceProvider(options =>
                    options.ValidateScopes = false)
                .Build();
    }
}
```

Конфигурирование ASP.NET Core подробно объясняется в главе 14, а просто отметим, что это единственное изменение, которое требуется для приложения `SportsStore`.

Создание миграции базы данных

Инфраструктура Entity Framework Core способна генерировать схему для базы данных, используя классы моделей, с помощью средства, которое называется *миграция*-ми. При подготовке миграции инфраструктура EF Core создает класс C#, содержащий команды SQL, которые нужны для подготовки базы данных. Если необходимо модифицировать классы моделей, тогда можно создать новую миграцию, которая содержит команды SQL, требуемые для отражения изменений. В результате не приходится беспокоиться о написании вручную и тестировании команд SQL и можно сосредоточиться на классах модели C# в приложении.

Команды Entity Framework Core выполняются из командной строки. Откроем окно командной строки или окно PowerShell, перейдем в папку проекта SportsStore (ту, что содержит файлы Startup.cs и appsettings.json) и запустим следующую команду, что-бы создать класс миграции, который подготовит базу данных к первому использованию:

```
dotnet ef migrations add Initial
```

Когда команда завершит свое выполнение, в окне Solution Explorer среды Visual Studio появится папка Migrations. Именно в ней инфраструктура Entity Framework Core хранит классы миграции. Одно из имен файлов будет выглядеть как отметка времени, за которой следует `_Initial.cs`, и в этом файле определен класс, применяемый для создания начальной схемы базы данных. Просмотрев содержимое такого файла, можно выяснить, каким образом класс модели `Product` использовался для создания схемы.

Что насчет команд Add-Migration и Update-Database?

Если вы имеете опыт разработки с помощью Entity Framework, то могли привыкнуть к применению команды `Add-Migration` для создания миграции базы данных и команды `Update-Database` для ее применения к базе данных.

С появлением платформы .NET Core в инфраструктуру Entity Framework Core были добавлены команды, интегрированные в инструмент командной строки `dotnet`, который использует пакет `Microsoft.EntityFrameworkCore.Tools.DotNet`, добавленный к проекту в листинге 8.14. Такие команды применяются в настоящей главе, поскольку они согласуются с другими командами .NET и могут использоваться в любом окне командной строки или PowerShell в отличие от команд `Add-Migration` и `Update-Database`, которые работают только в специальном окне Visual Studio.

Создание начальных данных

Чтобы заполнить базу данных и предоставить какие-то тестовые данные, добавим в папку `Models` файл класса по имени `SeedData.cs` с определением, приведенным в листинге 8.20.

Листинг 8.20. Содержимое файла `SeedData.cs` из папки `Models`

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
namespace SportsStore.Models {
```



```

public static class SeedData {
    public static void EnsurePopulated(IApplicationBuilder app) {
        ApplicationDbContext context = app.ApplicationServices
            .GetRequiredService<ApplicationDbContext>();
        context.Database.Migrate();
        if (!context.Products.Any()) {
            context.Products.AddRange(
                new Product {
                    Name = "Kayak",
                    Description = "A boat for one person",
                    Category = "Watersports", Price = 275 },
                new Product {
                    Name = "Lifejacket",
                    Description = "Protective and fashionable",
                    Category = "Watersports", Price = 48.95m },
                new Product {
                    Name = "Soccer Ball",
                    Description = "FIFA-approved size and weight",
                    Category = "Soccer", Price = 19.50m },
                new Product {
                    Name = "Corner Flags",
                    Description = "Give your playing field a professional touch",
                    Category = "Soccer", Price = 34.95m },
                new Product {
                    Name = "Stadium",
                    Description = "Flat-packed 35,000-seat stadium",
                    Category = "Soccer", Price = 79500 },
                new Product {
                    Name = "Thinking Cap",
                    Description = "Improve brain efficiency by 75%",
                    Category = "Chess", Price = 16 },
                new Product {
                    Name = "Unsteady Chair",
                    Description = "Secretly give your opponent a disadvantage",
                    Category = "Chess", Price = 29.95m },
                new Product {
                    Name = "Human Chess Board",
                    Description = "A fun game for the family",
                    Category = "Chess", Price = 75 },
                new Product {
                    Name = "Bling-Bling King",
                    Description = "Gold-plated, diamond-studded King",
                    Category = "Chess", Price = 1200
                }
            );
            context.SaveChanges();
        }
    }
}

```

Статический метод `EnsurePopulated()` получает аргумент типа `IApplicationBuilder`, представляющий собой интерфейс, который применяется в методе `Configure()` класса `Startup` для регистрации компонентов промежуточного про-

граммного обеспечения, обрабатывающих HTTP-запросы, и именно здесь будет обеспечено наличие в базе данных содержимого.

Метод `EnsurePopulated()` получает объект `ApplicationDbContext` через интерфейс `IApplicationBuilder` и вызывает метод `Database.Migrate()`, чтобы гарантировать применение миграции, что означает создание и подготовку базы данных к хранению объектов `Product`. Далее проверяется количество объектов `Product` в базе данных. Если объекты в базе данных отсутствуют, то база данных заполняется с использованием коллекции объектов `Product` посредством метода `AddRange()` и затем записывается с помощью метода `SaveChanges()`.

Финальное изменение заключается в заполнении базы данных начальной информацией при запуске приложения, для чего в класс `Startup` добавляется вызов метода `EnsurePopulated()`, как показано в листинге 8.21.

Листинг 8.21. Заполнение базы данных начальной информацией 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();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

В результате запуска приложения база данных создается, заполняется и применяется для обеспечения приложения своими данными. (Запаситесь терпением; создание базы данных может занять некоторое время).

Когда браузер запрашивает стандартный URL для приложения, конфигурация приложения сообщает MVC о необходимости создания контроллера `Product` для обработки запроса. Создание контроллера `Product` означает вызов конструктора класса `ProductController`, которому требуется объект, реализующий интерфейс `IProductRepository`, и новая конфигурация указывает MVC о том, что для этого должен быть создан и применен объект `EFProductRepository`. Объект `EFProductRepository` обращается к функциональности Entity Framework Core, которая загружает данные из SQL Server и преобразует их в объекты `Product`. Вся упомянутая работа скрыта от класса `ProductController`, который просто получает объект, реализующий интерфейс `IProductRepository`, и пользуется данными, которые он предоставляет. В итоге окно браузера отображает тестовую информацию из базы данных (рис. 8.8).

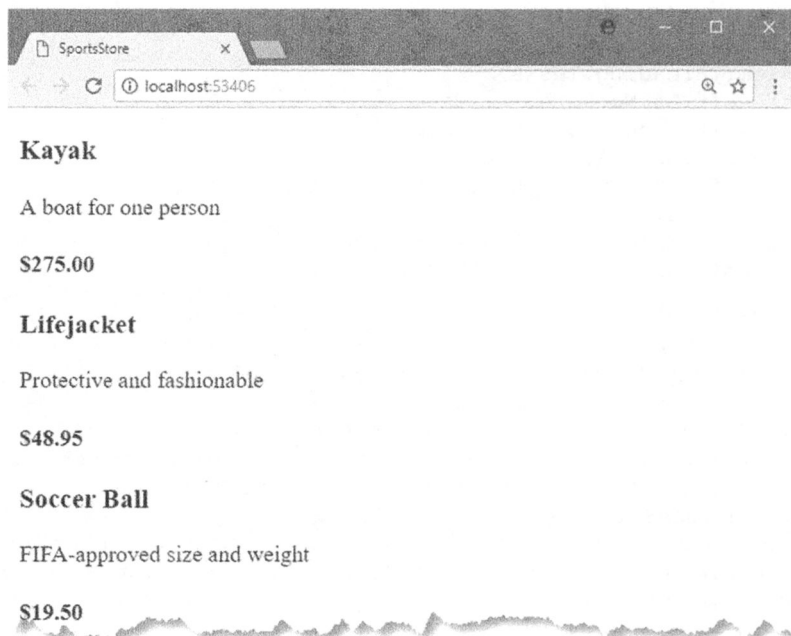


Рис. 8.8. Использование хранилища в виде базы данных

Такой подход с применением Entity Framework Core для представления базы данных SQL Server в виде последовательности объектов моделей отличается простотой и легкостью, позволяя сосредоточить внимание на инфраструктуре ASP.NET Core MVC. Я опустил множество деталей, касающихся оперирования EF Core, и большое количество доступных конфигурационных параметров. Мне нравится инфраструктура Entity Framework Core, и я рекомендую уделить время на ее изучение. Хорошей отправной точкой послужит веб-сайт Microsoft для Entity Framework Core (<https://docs.microsoft.com/ru-ru/ef/#pivot=efcore>) или моя будущая книга по Entity Framework Core, которая выйдет в издательстве Apress.

Добавление поддержки разбиения на страницы

На рис. 8.8 видно, что представление `List.cshtml` отображает сведения о товарах в базе данных на одной странице. В этом разделе мы добавим поддержку разбиения на страницы, чтобы представление отображало на странице меньшее число товаров, а пользователь мог переходить со страницы на страницу для просмотра всего каталога. Для решения задачи добавим в метод действия `List()` контроллера `Product` параметр (листинг 8.22).

Листинг 8.22. Добавление поддержки разбиения на страницы в файле `ProductController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

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

        public IActionResult List(int productPage = 1)
            => View(repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((productPage - 1) * PageSize)
                .Take(PageSize));
    }
}
```

Поле `PageSize` указывает, что на одной странице должны отображаться сведения о четырех товарах. В метод `List()` добавлен необязательный параметр, который означает, что в случае вызова метода без параметра (`List()`) вызов трактуется так, словно ему было передано значение, указанное в определении параметра (`List(1)`). В результате метод действия отображает первую страницу сведений о товарах, когда инфраструктура MVC вызывает его без аргумента. Внутри тела метода действия мы получаем объекты `Product`, упорядочиваем их по первичному ключу, пропускаем товары, которые располагаются до начала текущей страницы, и выбираем количество товаров, указанное в поле `PageSize`.

Модульное тестирование: разбиение на страницы

Модульное тестирование средства разбиения на страницы можно провести, создав имитированное хранилище, внедрив его в конструктор класса `ProductController` и вызвав метод `List()`, чтобы запросить конкретную страницу. Затем полученные объекты `Product` можно сравнить с теми, которые ожидалось от тестовых данных в имитированной реализации. Написание модульных тестов обсуждалось в главе 7. Ниже показан созданный для этой цели модульный тест, который находится в файле класса по имени `ProductControllerTests.cs`, добавленном в проект `SportsStore.Tests`:

```

using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {
    public class ProductControllerTests {
        [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
            IEnumerable<Product> result =
                controller.List(2).ViewData.Model as IEnumerable<Product>;

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

```

Получение данных, возвращаемых из метода действия, выглядит несколько неуклюже. Результатом является объект `ViewResult`, и значение его свойства `ViewData.Model` должно быть приведено к ожидаемому типу данных. В главе 17 рассматриваются разнообразные результирующие типы, которые могут возвращать методы действий, а также способы работы с ними.

Отображение ссылок на страницы

Запустив приложение, вы увидите, что теперь на странице отображаются четыре позиции каталога. Если нужно просмотреть другую страницу, тогда в конец URL можно добавить параметр строки запроса, например:

```
http://localhost:5000/?productPage=2
```

Вы должны указать в URL номер порта, который был назначен вашему проекту. Используя такие строки запросов, можно перемещаться по каталогу товаров.

У пользователей нет какого-либо способа выяснить о существовании таких параметров строки запроса, но даже если бы он был, то вряд ли бы они захотели иметь

дело с навигацией подобного вида. Взамен мы должны визуализировать в нижней части каждого списка товаров ссылки на страницы, чтобы пользователи могли переходить между страницами. Мы реализуем вспомогательную функцию дескриптора, которая будет генерировать HTML-разметку для требуемых ссылок.

Добавление модели представления

Чтобы обеспечить поддержку вспомогательной функции дескриптора, мы собираемся передавать представлению информацию о количестве доступных страниц, текущей странице и общем числе товаров в хранилище. Проще всего это сделать, создав класс модели представления, который специально применяется для передачи данных между контроллером и представлением. Создадим в проекте SportsStore папку Models/ViewModels и добавим в нее файл класса с содержимым, приведенным в листинге 8.23.

Листинг 8.23. Содержимое файла PagingInfo.cs из папки Models/ViewModels

```
using System;

namespace SportsStore.Models.ViewModels {
    public class PagingInfo {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages =>
            (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
    }
}
```

Добавление класса вспомогательной функции дескриптора

Теперь, имея модель представления, можно создать вспомогательную функцию дескриптора. Создадим в проекте SportsStore папку Infrastructure и добавим в нее файл класса по имени PageLinkTagHelper.cs с определением, показанным в листинге 8.24. Вспомогательные функции дескрипторов являются крупной частью инфраструктуры ASP.NET Core MVC, и в главах 23–25 объясняется, как они работают и каким образом их создавать.

Совет. В папку Infrastructure будут помещаться классы, которые предоставляют приложению связующий код, но не имеют отношения к предметной области приложения.

Листинг 8.24. Содержимое файла 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;
```

```

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; }

        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");
                tag.Attributes["href"] = urlHelper.Action(PageAction,
                    new { productPage = i });
                tag.InnerHtml.Append(i.ToString());
                result.InnerHtml.AppendHtml(tag);
            }
            output.Content.AppendHtml(result.InnerHtml);
        }
    }
}

```

Класс вспомогательной функции дескриптора `PageLinkTagHelper` заполняет элемент `div` элементами `a`, которые соответствуют страницам товаров. Сейчас мы не будем вдаваться в какие-либо детали относительно вспомогательных функций дескрипторов; достаточно знать, что они предлагают один из наиболее удобных способов помещения логики C# в представления.

Код для вспомогательной функции дескриптора может выглядеть запутанным, потому что смешивать C# и HTML непросто. Но использование вспомогательных функций дескрипторов является предпочтительным способом включения блоков кода C# в представление, поскольку вспомогательную функцию дескриптора можно легко подвергать модульному тестированию.

Большинство компонентов MVC, таких как контроллеры и представления, обнаруживаются автоматически, но вспомогательные функции дескрипторов должны быть зарегистрированы.

В листинге 8.25 приведено содержимое файла `_ViewImports.cshtml` из папки `Views` с добавленным оператором, который сообщает MVC о том, что вспомогательные функции дескрипторов следует искать в пространстве имен `SportsStore.Infrastructure`. Кроме того, добавлено также выражение `@using`, чтобы на классы моделей представлений можно было ссылаться в представлениях, не указывая пространство имен.

Листинг 8.25. Регистрация вспомогательной функции дескриптора в файле `_ViewImports.cshtml` из папки `Views/Home`

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

Модульное тестирование: создание ссылок на страницы

Чтобы протестировать класс вспомогательной функции дескриптора `PageLinkTagHelper`, вызывается метод `Process()` с тестовыми данными и предоставляется объект `TagHelperOutput`, который инспектируется на предмет сгенерированной HTML-разметки. Тест определен в новом файле `PageLinkTagHelperTests.cs` внутри проекта `SportsStore.Tests`:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests {
    public class PageLinkTagHelperTests {
        [Fact]
        public void Can_Generate_Page_Links() {
            // Организация
            var urlHelper = new Mock<IUrlHelper>();
            urlHelper.SetupSequence(x => x.Action(It.IsAny<UrlActionContext>()))
                .Returns("Test/Page1")
                .Returns("Test/Page2")
                .Returns("Test/Page3");

            var urlHelperFactory = new Mock<IUrlHelperFactory>();
            urlHelperFactory.Setup(f =>
                f.GetUrlHelper(It.IsAny<ActionContext>()))
                .Returns(urlHelper.Object);

            PageLinkTagHelper helper =
                new PageLinkTagHelper(urlHelperFactory.Object) {
                    PageModel = new PagingInfo {
                        CurrentPage = 2,
                        TotalItems = 28,
                        ItemsPerPage = 10
                    },
                    PageAction = "Test"
                };

            TagHelperContext ctx = new TagHelperContext(
                new TagHelperAttributeList(),
                new Dictionary<object, object>(), "");
```



```

var content = new Mock<TagHelperContent>();
TagHelperOutput output = new TagHelperOutput("div",
    new TagHelperAttributeList(),
    (cache, encoder) => Task.FromResult(content.Object));
// Действие
helper.Process(ctx, output);
// Утверждение
Assert.Equal(@"<a href=""Test/Page1"">1</a>"
    + @"<a href=""Test/Page2"">2</a>"
    + @"<a href=""Test/Page3"">3</a>",
    output.Content.GetContent());
    }
}
}

```

Сложность теста связана с построением объектов, которые требуются для создания и применения вспомогательной функции дескриптора. Вспомогательные функции дескрипторов используют объекты реализации `IUrlHelperFactory` для генерации URL, которые указывают на разные части приложения. Для создания реализации интерфейса `IUrlHelperFactory` и связанного с ним `IUrlHelper`, предоставляющего тестовые данные, используется инфраструктура `Moq`.

Основная часть теста проверяет вывод вспомогательной функции дескриптора с применением литерального строкового значения, которое содержит двойные кавычки. Язык C# позволяет работать с такими строками при условии, что строка предварена символом `@`, а вместо одной двойной кавычки внутри строки используется набор из двух двойных кавычек (`""`). Вы должны помнить о том, что разносить литеральную строку по нескольким строкам файла нельзя, если только строка, с которой производится сравнение, не разнесена аналогичным образом. Например, литерал, применяемый в тестовом методе, был размещен в нескольких строках из-за недостаточной ширины печатной страницы. Символ новой строки не добавлялся, иначе тест не прошел бы.

Добавление данных модели представления

Мы пока не готовы использовать вспомогательную функцию дескриптора, т.к. представление еще не снабжено экземпляром класса модели представления `PagingInfo`. Это можно было бы сделать с применением объекта `ViewBag`, но лучше поместить все данные, подлежащие передаче из контроллера в представление, внутрь единственного класса модели представления. Добавим в папку `Models/ViewModels` проекта `SportsStore` файл класса по имени `ProductsListViewModel.cs` с содержимым, приведенным в листинге 8.26.

Листинг 8.26. Содержимое файла `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; }
    }
}

```

Далее можно обновить метод действия `List()` класса `ProductController`, чтобы он использовал класс `ProductsListViewModel` для снабжения представления сведениями о товарах, отображаемых на страницах, и информацией о разбиении на страницы (листинг 8.27).

**Листинг 8.27. Обновление метода 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 IActionResult List(int productPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            });
    }
}
```

Внесенные изменения обеспечивают передачу представлению объекта `ProductsListViewModel` как данных модели.

**Модульное тестирование:
данные разбиения на страницы для модели представления**

Нам необходимо удостовериться в том, что контроллер отправляет представлению корректную информацию о разбиении на страницы. Ниже показан модульный тест, добавленный в класс `ProductControllerTests` внутри тестового проекта, который обеспечивает такую проверку:

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

```

Потребуется также модифицировать ранее созданный модульный тест для проверки разбиения на страницы, содержащийся в методе `Can_Paginate()`. Он полагается на метод действия `List()`, возвращающий объект `ViewResult`, свойством `Model` которого является последовательность объектов `Product`, но мы поместили эти данные внутрь еще одного типа модели представления. Вот переделанный тест:

```

...
[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(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);
}
...

```

Учитывая объем дублированного кода в двух тестовых методах, обычно следовало бы создать общий метод начальной установки. Однако, поскольку модульные тесты описаны в индивидуальных врезках вроде текущей, их код представлен по отдельности, чтобы с каждым тестом можно было ознакомиться независимо от других.

В настоящий момент представление ожидает последовательность объектов `Product`, поэтому файл `List.cshtml` необходимо обновить, как показано в листинге 8.28, чтобы иметь дело с новым типом модели представления.

Листинг 8.28. Обновление файла `List.cshtml` из папки `Views/Product`

```
@model ProductsListViewModel
@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

Выражение `@model` было изменено для указания механизму Razor на то, что теперь мы работаем с другим типом данных. Цикл `foreach` был обновлен, чтобы источником данных стало свойство `Products` объекта модели.

Отображение ссылок на страницы

Наконец-то все готово для добавления в представление `List` ссылок на страницы. Мы создали модель представления, которая содержит информацию о разбиении на страницы, обновили контроллер, чтобы эта информация передавалась представлению, и модифицировали выражение `@model`, приведя его в соответствие с новым типом модели представления. Осталось добавить HTML-элемент, который вспомогательная функция дескриптора будет обрабатывать для создания ссылок на страницы (листинг 8.29).

Листинг 8.29. Добавление ссылок на страницы в файле `List.cshtml` из папки `Views/Product`

```
@model ProductsListViewModel
@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
<div page-model="@Model.PagingInfo" page-action="List"></div>
```

Запустив приложение, вы увидите новые ссылки на страницы (рис. 8.9). Стиль отображения довольно примитивен, но далее в главе он будет улучшен. Пока важно то, что ссылки позволяют перемещаться по страницам каталога и знакомиться с товарами, выставленными на продажу. Когда механизм Razor обнаруживает атрибут `page-model` в элементе `div`, он обращается к классу `PageLinkTagHelper` для преобразования элемента, что и дает набор ссылок, показанных на рис. 8.9.

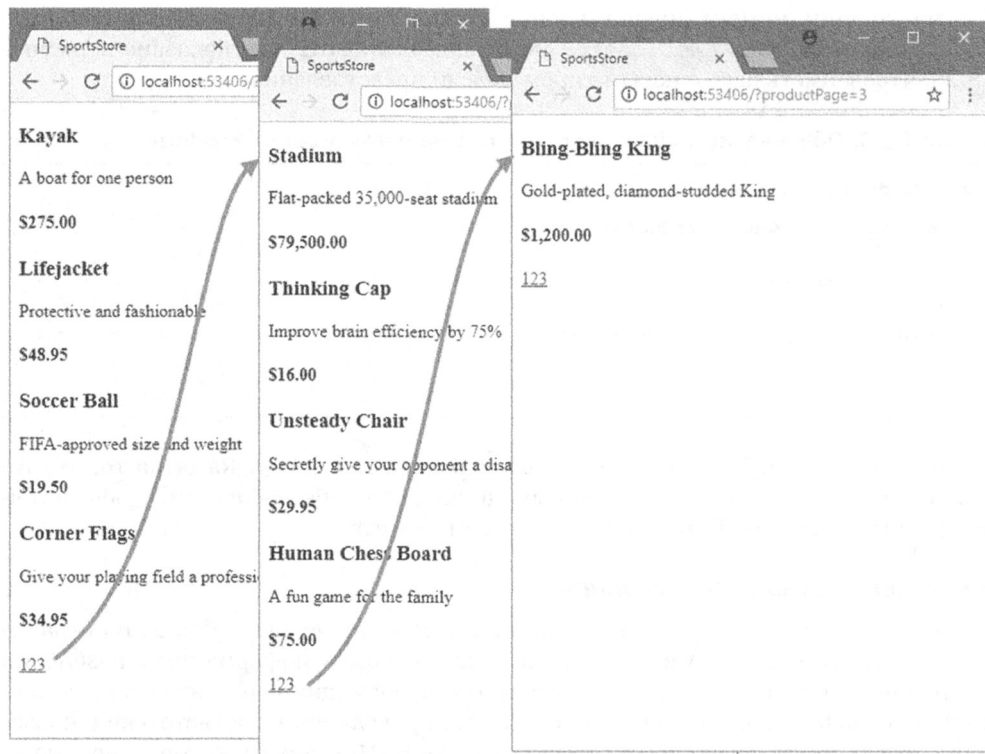


Рис. 8.9. Отображение ссылок для навигации по страницам

Почему бы просто не воспользоваться элементом управления GridView?

Если вы ранее работали с ASP.NET, то вам может показаться, что такой большой объем работы привел к довольно скромным результатам. Пришлось написать немало кода лишь для того, чтобы получить список товаров с разбиением на страницы. Если бы мы прибегли к услугам инфраструктуры Web Forms, то такого же результата можно было бы достичь с применением готового элемента управления GridView или ListView из ASP.NET Web Forms, привязав его напрямую к таблице Products базы данных.

То, что было сделано в главе, выглядит не слишком впечатляюще, но серьезно отличается от процесса перетаскивания какого-то элемента управления на поверхность визуального проектирования. Во-первых, мы строим приложение с надежной и удобной в сопровождении архитектурой, которая обеспечивает надлежащее разделение обязанностей. В отличие от простейшего использования ListView мы не связываем напрямую пользовательский интерфейс и базу данных, что представляет собой подход, который дает быстрые результаты, но вызовет проблемы и сложности в будущем. Во-вторых, в ходе дела мы создаем модульные тесты, что позволяет проверять поведение приложения естественным образом, который практически невозможен в случае применения сложного элемента управления Web Forms. В-третьих, не забывайте, что значительная часть главы посвящена созданию инфраструктуры, на основе которой строится приложение. Например, определить и реализовать хранилище нужно только один раз, и теперь, когда оно имеется в нашем распоряжении, новые функциональные средства можно строить и тестировать легко и быстро, как будет продемонстрировано в последующих главах.

Разумеется, ничто из сказанного отнюдь не умаляет значимость безотлагательных результатов, которые способна предложить инфраструктура Web Forms, но, как объяснялось в главе 3, такая безотлагательность имеет свою цену, которая может оказаться высокой и болезненной в крупных и сложных проектах.

Улучшение URL

Ссылки на страницы работают, но для передачи информации серверу они по-прежнему используют строку запроса следующего вида:

```
http://localhost/?productPage=2
```

Чтобы получить более привлекательные URL, необходимо создать схему, которая следует шаблону *компоуемых URL*. Компонуемым называется URL, имеющий смысл для пользователя, такой как показанный ниже:

```
http://localhost/Page2
```

Инфраструктура MVC позволяет легко изменять схему URL в приложении, потому что применяет средство *маршрутизации* ASP.NET, которое отвечает за обработку URL для выяснения, на какую часть приложения они указывают. Понадобится лишь добавить новый маршрут при регистрации промежуточного программного обеспечения MVC в методе `Configure()` класса `Startup` (листинг 8.30).

Листинг 8.30. Добавление нового маршрута 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 Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

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();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
        }
    }
}
```

```

app.UseStatusCodePages();
app.UseStaticFiles();
app.UseMvc(routes => {
    routes.MapRoute(
        name: "pagination",
        template: "Products/Page{productPage}",
        defaults: new { Controller = "Product", action = "List" });
    routes.MapRoute(
        name: "default",
        template: "{controller=Product}/{action=List}/{id?}");
});
SeedData.EnsurePopulated(app);
}
}
}

```

Новый маршрут важно поместить перед стандартным маршрутом (по имени `default`), который уже присутствует в методе. Как будет показано в главе 15, система маршрутизации обрабатывает маршруты в порядке их перечисления, а нам нужно, чтобы новый маршрут имел преимущество перед существующим маршрутом.

Это единственное изменение, которое требуется внести, чтобы изменить схему URL для разбиения на страницы списка товаров. Инфраструктура MVC тесно интегрирована с функцией маршрутизации, а потому приложение автоматически отражает изменение подобного рода в URL, используемых приложением, включая URL, которые генерируются вспомогательными функциями дескрипторов вроде применяемых для генерации ссылок на страницы. Не беспокойтесь, если маршрутизация пока не особенно понятна — она будет подробно рассматриваться в главах 15 и 16.

Запустив приложение и щелкнув на ссылке для какой-нибудь страницы, можно наблюдать новую схему URL в действии (рис. 8.10).

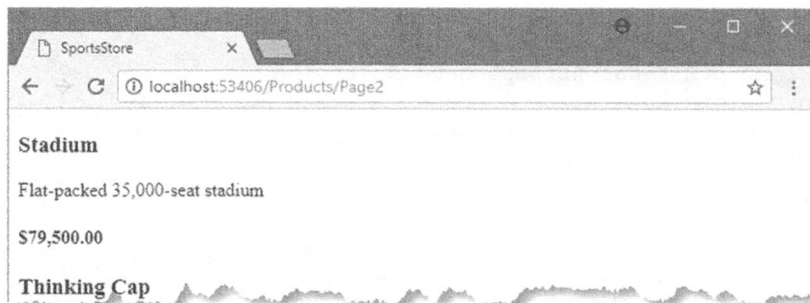


Рис. 8.10. Новая схема URL, отображаемая в браузере

Стилизация содержимого

Мы построили значительную часть инфраструктуры и базовые средства приложения и готовы к сборке всего воедино, но пока совершенно не уделяли внимание его внешнему виду. Хотя данная книга не посвящена веб-дизайну или CSS, внешний вид приложения `SportsStore` настолько примитивен, что умаляет его технические достоинства. В настоящем разделе мы постараемся исправить ситуацию. Мы собираемся реализовать классическую компоновку, содержащую два столбца и заголовок (рис. 8.11).

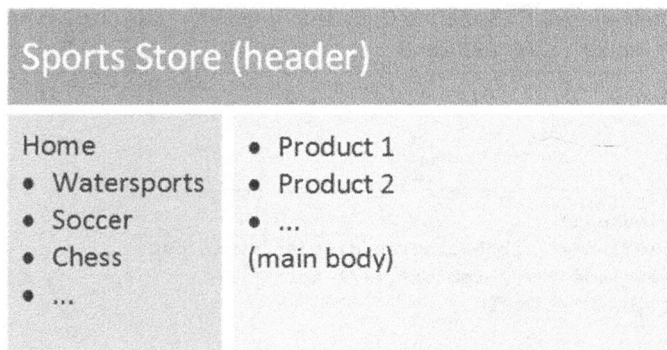


Рис. 8.11. Целевой дизайн для приложения SportsStore

Установка пакета Bootstrap

Для предоставления стилей CSS, которые будут применены к приложению, мы планируем использовать пакет Bootstrap. При установке пакета Bootstrap мы будем полагаться на встроенную в Visual Studio поддержку инструмента Bower. Выберем шаблон элемента Bower Configuration File (Файл конфигурации Bower) из категории General в диалоговом окне Add New Item, чтобы создать в проекте SportsStore файл по имени `bower.json`, как демонстрировалось в главе 6. Добавим в раздел `dependencies` созданного файла пакет Bootstrap, как показано в листинге 8.31. Ранее уже упоминалось, что в примерах книги применяется предварительный выпуск Bootstrap.

Листинг 8.31. Добавление пакета Bootstrap в файле `bower.json` из папки SportsStore

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

После сохранения изменений, внесенных в файл `bower.json`, среда Visual Studio с помощью инструмента Bower загрузит пакет Bootstrap в папку `wwwroot/lib/bootstrap`. Пакет Bootstrap зависит от пакета jQuery, который также автоматически добавится в проект.

Применение стилей Bootstrap к компоновке

В главе 5 объяснялось, как работают представления Razor, как они используются и как встраивать в них компоновки. Файл запуска представления, добавленный в начале главы, указывает, что в качестве стандартной компоновки должен выступать файл по имени `_Layout.cshtml`, так что начальную стилизацию Bootstrap мы применим именно к нему (листинг 8.32).

Листинг 8.32. Применение CSS-файла Bootstrap к файлу `_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">
      Put something useful here later
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Элемент `link` имеет атрибуты `asp-href-include` и `asp-href-exclude`, которые представляют собой пример использования встроенного класса вспомогательной функции дескриптора. В этом случае вспомогательная функция дескриптора просматривает значения атрибутов и генерирует элементы `link` для всех файлов по пути, указанному атрибутом `include`, и по путям, указанным атрибутами `exclude`. Пути в атрибутах могут содержать групповые символы, что делает такой прием удобным способом обеспечения возможности добавления и удаления файлов из структуры папок `wwwroot`, не нарушая работу приложения. Но, как объясняется в главе 25, следует проявлять внимание, чтобы из указанных путей выбирались только те файлы, которые ожидается получить.

Добавление в компоновку таблицы стилей CSS из Bootstrap означает, что определенные в ней стили можно применять в любом представлении, которое полагается на данную компоновку. В листинге 8.33 стилизация используется в файле `List.cshtml`.

Листинг 8.33. Стилизация содержимого в файле `List.cshtml` из папки `Views/Product`

```
@model ProductsListViewModel
@foreach (var p in Model.Products) {
  <div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
      <h4>
        @p.Name
        <span class="badge badge-pill badge-primary" style="float:right">
```

```

        <small>@p.Price.ToString("c")</small>
    </span>
</h4>
</div>
<div class="card-text p-1">@p.Description</div>
</div>
}

<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" class="btn-group pull-right m-1">
</div>

```

Нам необходимо стилизовать кнопки, которые генерирует класс `PageLinkTagHelper`, но жестко встраивать классы `Bootstrap` в код нежелательно, т.к. в итоге затруднится повторное использование вспомогательной функции дескриптора где-то в другом месте приложения либо изменение внешнего вида кнопок. Взамен мы определяем в элементе `div` специальные атрибуты, которые указывают требуемые классы и соответствуют свойствам, добавленным во вспомогательную функцию дескриптора. Затем они применяются для стилизации создаваемых элементов а (листинг 8.34).

Листинг 8.34. Добавление классов к генерируемым элементам 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;

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; }

        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");
    tag.Attributes["href"] = urlHelper.Action(PageAction,
        new { productPage = i });
    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);
}
}
}

```

Значения атрибутов автоматически используются для установки значений свойств во вспомогательной функции дескриптора, учитывая отображение между форматом имени атрибута HTML (`page-class-normal`) и форматом имени свойства C# (`PageClassNormal`). Это позволяет вспомогательным функциям дескрипторов по-разному реагировать в зависимости от атрибутов HTML-элемента, обеспечивая более гибкий способ генерации содержимого в приложении MVC.

Запустив приложение, легко заметить, что его внешний вид улучшился — во всяком случае, в какой-то степени (рис. 8.12).

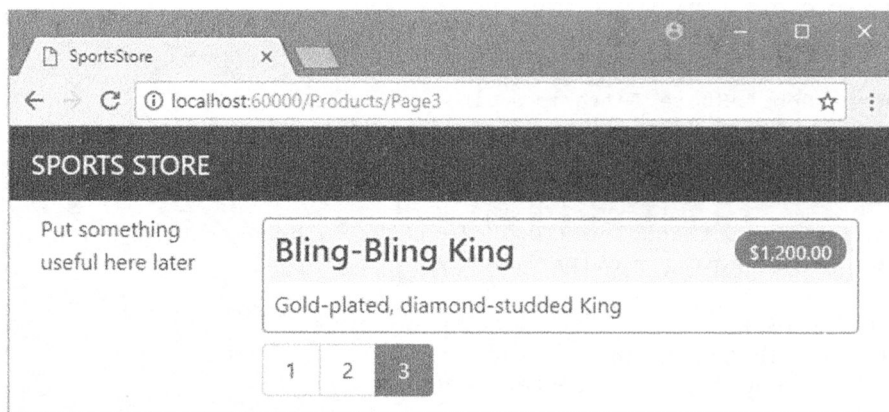


Рис. 8.12. Приложение SportsStore с улучшенным дизайном

Создание частичного представления

В качестве завершающего штриха в главе мы проведем рефакторинг приложения, чтобы упростить представление `List.cshtml`. Мы создадим *частичное представление*, являющееся фрагментом содержимого, которое можно внедрять в другое представление подобно шаблону. Частичные представления подробно рассматриваются в главе 21. Они помогают сократить дублирование, когда одно и то же содержимое

должно появляться в разных местах приложения. Вместо того чтобы копировать и вставлять одинаковую разметку Razor во множество представлений, можно определить ее единожды в частичном представлении. Чтобы создать частичное представление, добавим в папку Views/Shared файл представления Razor по имени ProductSummary.cshtml и поместим в него разметку, показанную в листинге 8.35.

Листинг 8.35. Содержимое файла 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>
  <div class="card-text p-1">@Model.Description</div>
</div>
```

Теперь необходимо модифицировать файл List.cshtml из папки Views/Product, чтобы в нем применялось частичное представление (листинг 8.36).

Листинг 8.36. Использование частичного представления в файле List.cshtml из папки Views/Product

```
@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" class="btn-group pull-right m-1">
</div>
```

Мы взяли код разметки, который ранее размещался в цикле foreach внутри представления List.cshtml, и перенесли его в новое частичное представление. Обращение к частичному представлению производится с помощью вспомогательного метода `Html.Partial()`, которому в аргументах передаются имя представления и объект модели представления. Подобного рода переход на частичное представление является рекомендуемым приемом, поскольку он позволяет вставлять одну и ту же разметку в любое представление, которое нуждается в отображении сводки о товаре. Как видно на рис. 8.13, добавление частичного представления не изменяет внешний вид приложения; оно просто меняет место, где механизм Razor находит содержимое, которое применяет для генерации ответа, отправляемого браузеру.

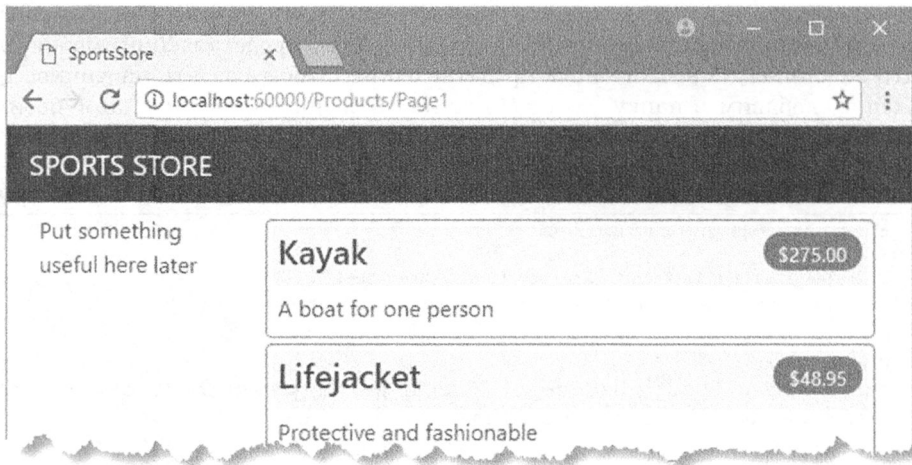


Рис. 8.13. Использование частичного представления

Резюме

В главе была построена основная инфраструктура для приложения SportsStore. Пока что она не содержит достаточного набора функциональных средств, чтобы их можно было продемонстрировать пользователю прямо сейчас, но “за кулисами” уже имеются зачатки модели предметной области с хранилищем товаров, которое поддерживается SQL Server и Entity Framework Core. В приложении присутствует единственный контроллер `Product`, который может создавать разбитые на страницы списки товаров, а также настроена ясная и дружелюбная к пользователям схема URL.

В настоящей главе могло показаться, что для получения весьма незначительных выгод пришлось провести излишне трудоемкую начальную подготовку, но в следующей главе равновесие будет восстановлено. Теперь, когда фундаментальная структура на месте, можно двигаться дальше и добавлять все средства, ориентированные на пользователя: навигацию по категориям, корзину для покупок и начальную форму для оплаты.

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 получают 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.

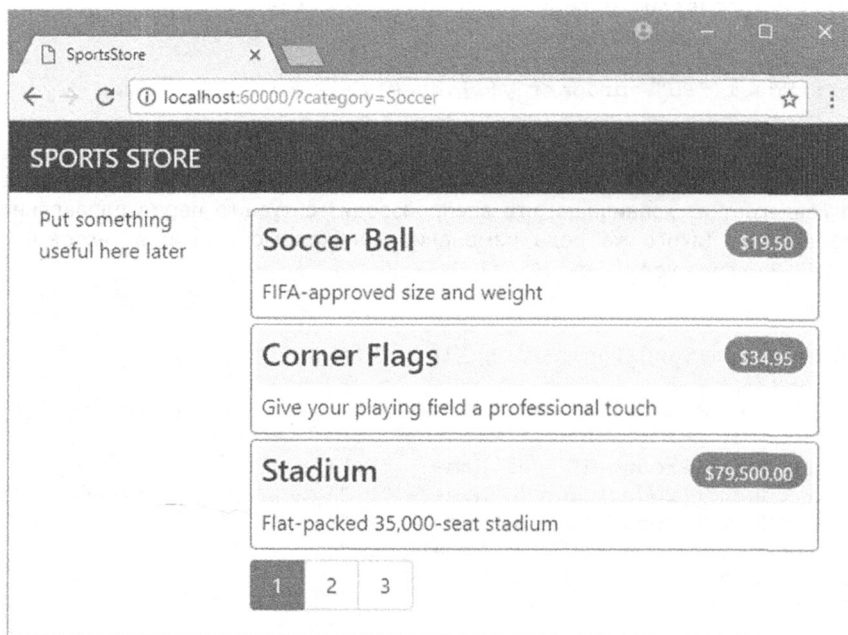


Рис. 9.1. Использование строки запроса для фильтрации по категории

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.

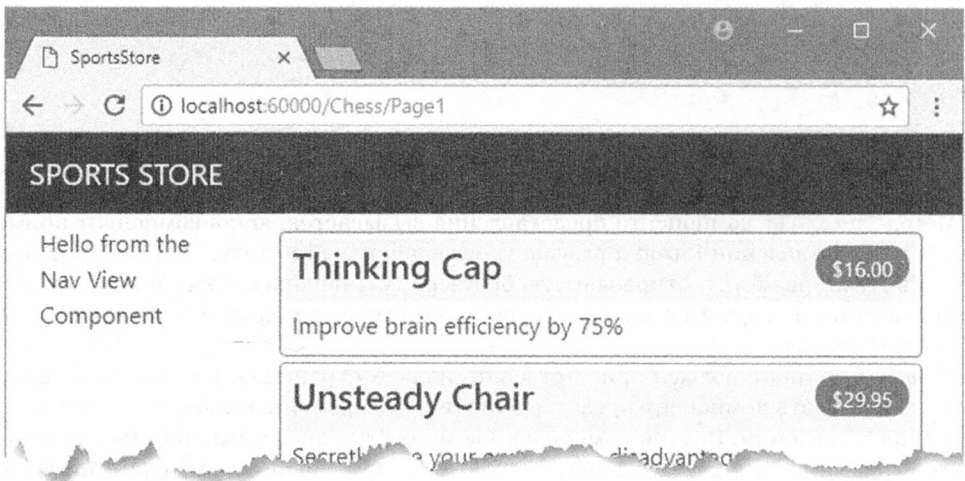
Листинг 9.7. Использование компонента представления в файле `_Layout.cshtml` из папки `Views/Shared`

```

<!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.

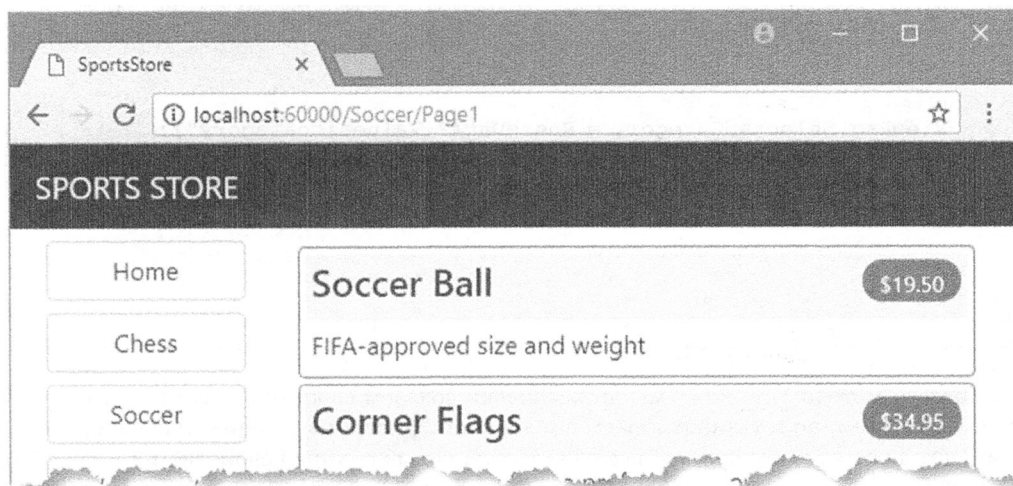


Рис. 9.3. Генерация ссылок на категории с помощью компонента представления

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 `NavigationMenuViewComponentTests`:

```
...
[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).

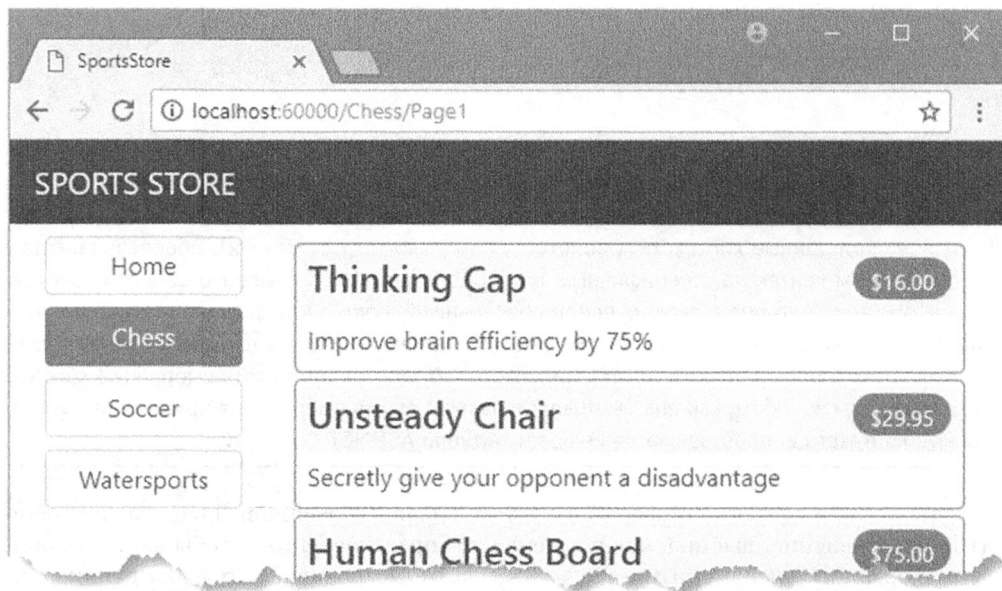


Рис. 9.4. Подсвечивание выбранной категории

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.

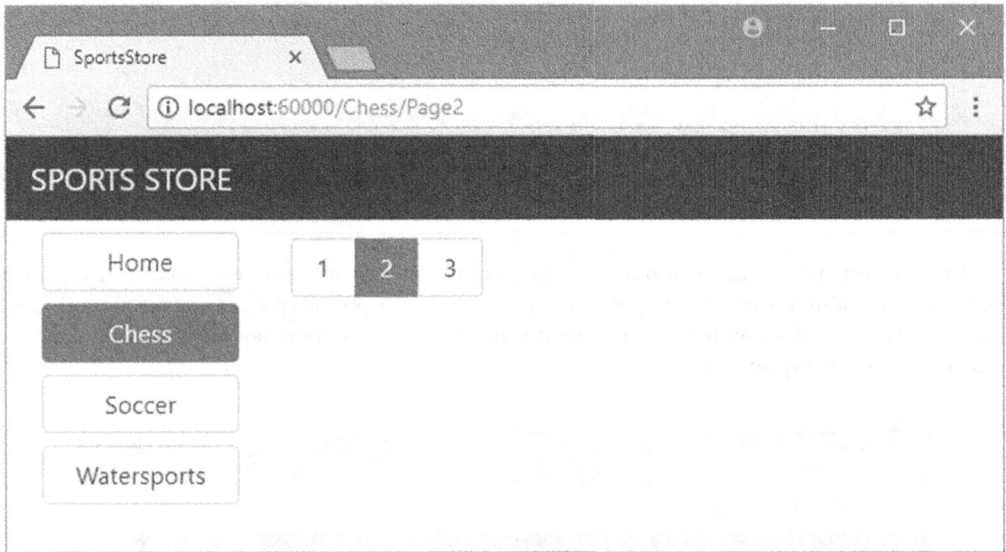


Рис. 9.5. Отображение некорректных ссылок на страницы, когда выбрана какая-то категория

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).

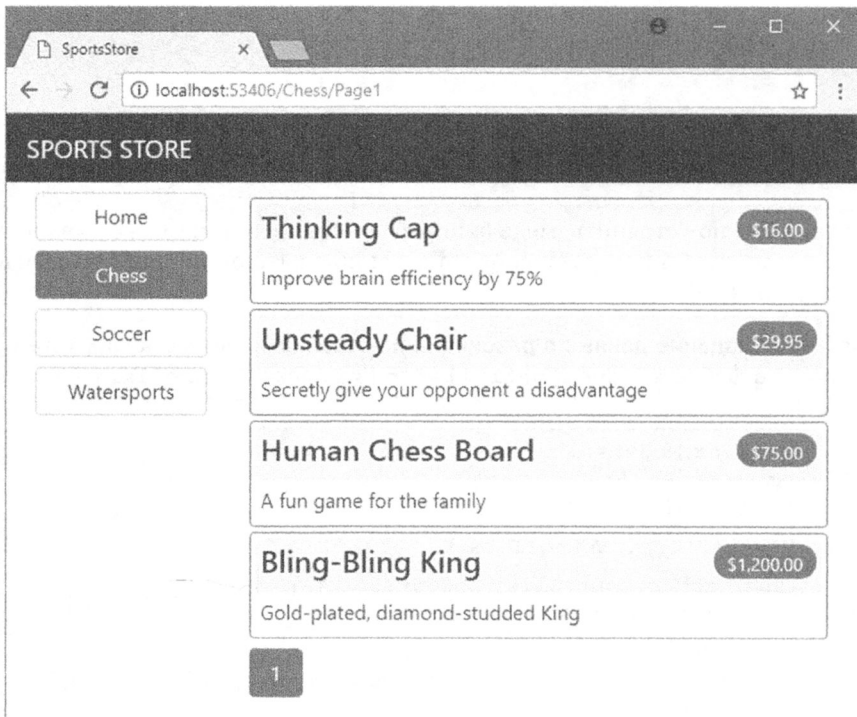


Рис. 9.6. Отображение ссылок на страницы с учетом выбранной категории

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.

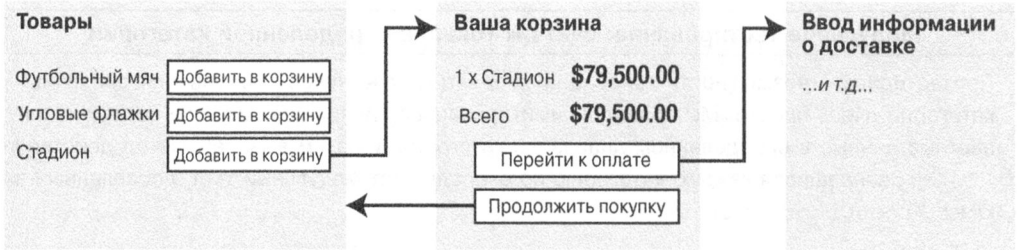


Рис. 9.7. Базовый поток корзины для покупок

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 управляют значением `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 сериализует 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:

```
...
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 `CartItemViewModel.cs` с содержимым, приведенным в листинге 9.20.

Листинг 9.20. Содержимое файла `CartItemViewModel.cs` из папки `Models/ViewModels`

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {
    public class CartItemViewModel {
        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>
</table>
```

```

<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.

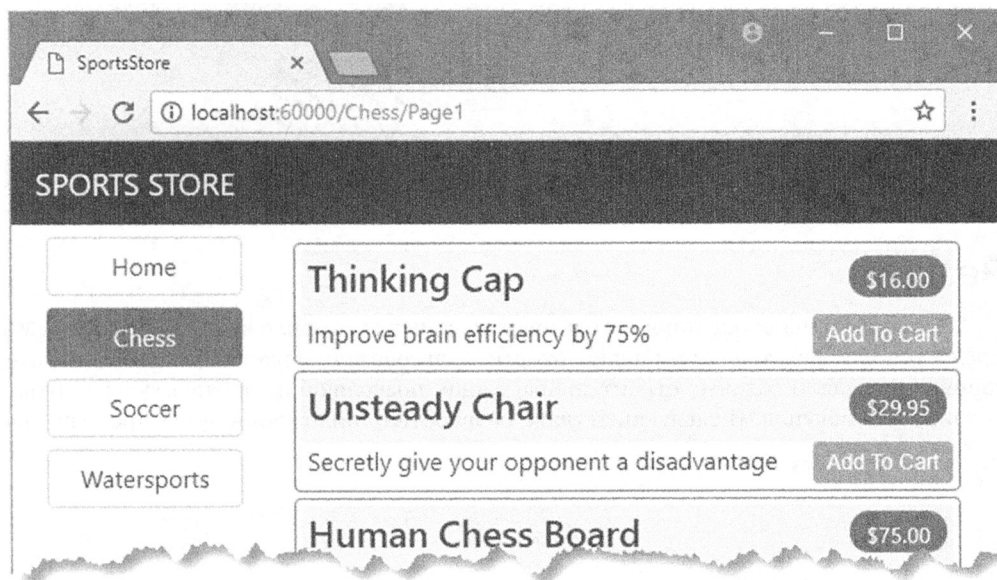


Рис. 9.8. Кнопки Add To Cart

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.

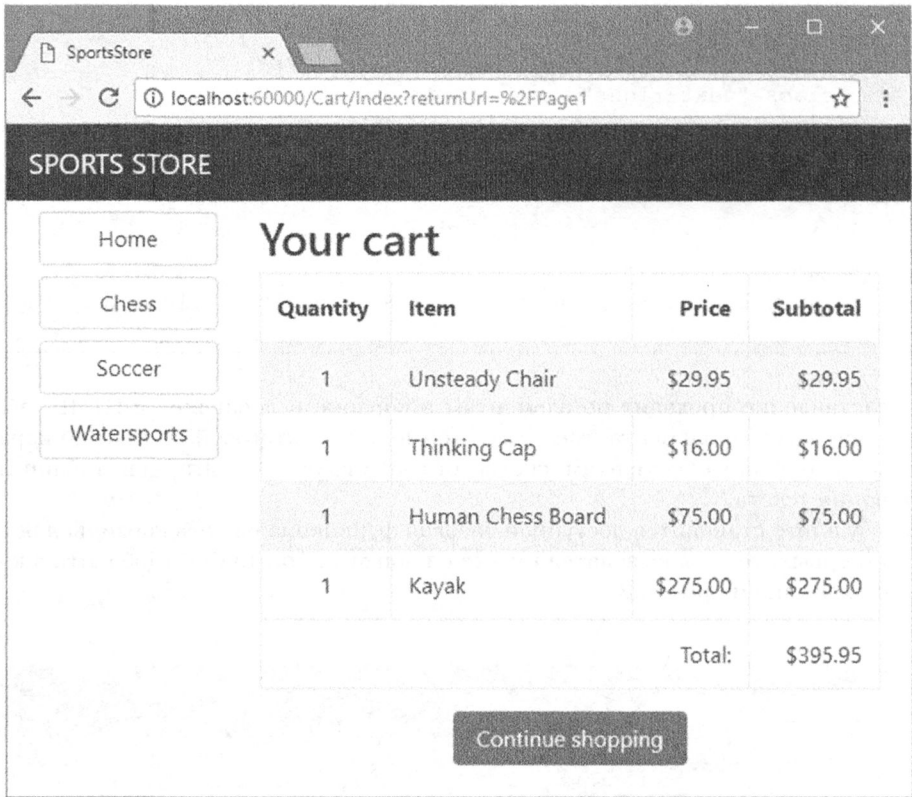


Рис. 9.9. Отображение содержимого корзины для покупок

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 сериализуют 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).

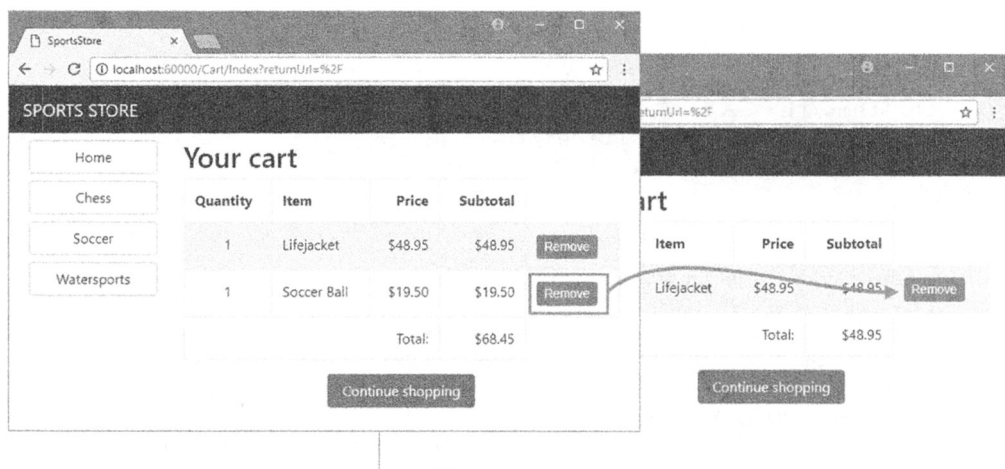


Рис. 10.1. Удаление элемента из корзины для покупок

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.

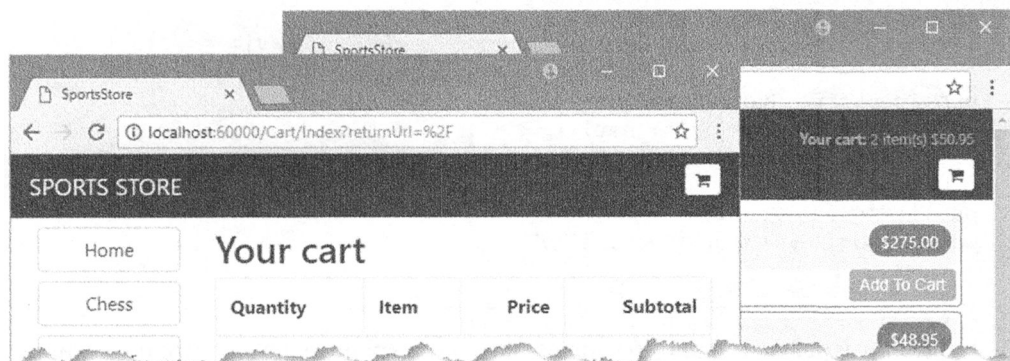


Рис. 10.2. Отображение итоговой информации по корзине

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`.

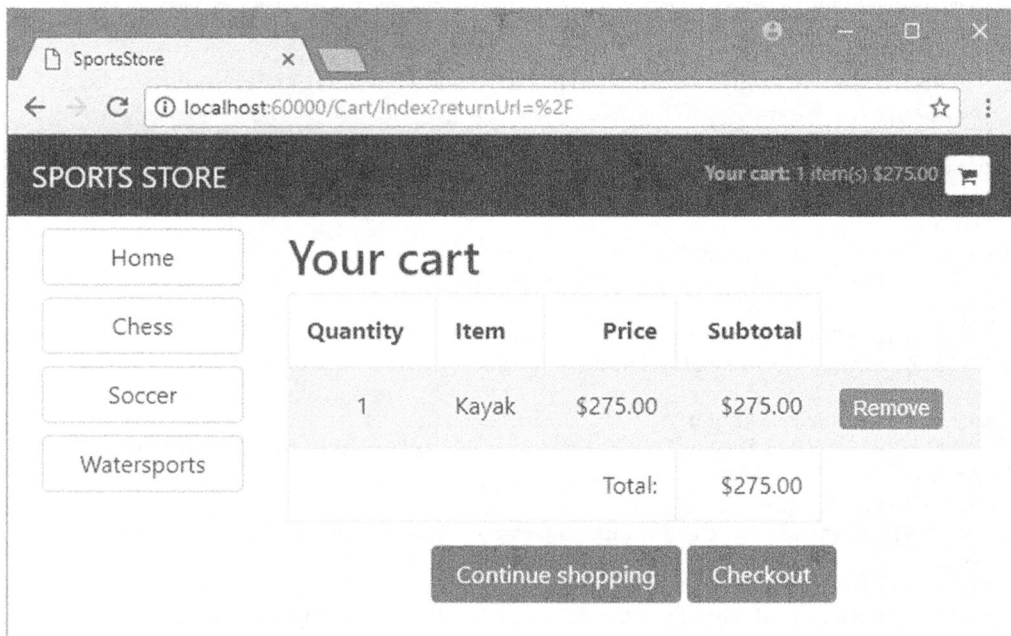


Рис. 10.3. Кнопка `Checkout`

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 IActionResult 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`.

SPORTS STORE

Your cart: 1 item(s) \$275.00

Home

Chess

Soccer

Watersports

Check out now

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

Ship to

Name:

Address

Line 1:

Line 2:

Line 3:

City:

State:

Zip:

Country:

Options

☐ Gift wrap these items

Complete Order

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 EOrderRepository.cs с определением, представленным в листинге 10.15.

Листинг 10.15. Содержимое файла EOrderRepository.cs из папки Models

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

namespace SportsStore.Models {
    public class EOrderRepository : IOrderRepository {
        private ApplicationDbContext context;

        public EOrderRepository(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 EOrderRepository 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 десериализованные 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 IActionResult 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 IActionResult 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.

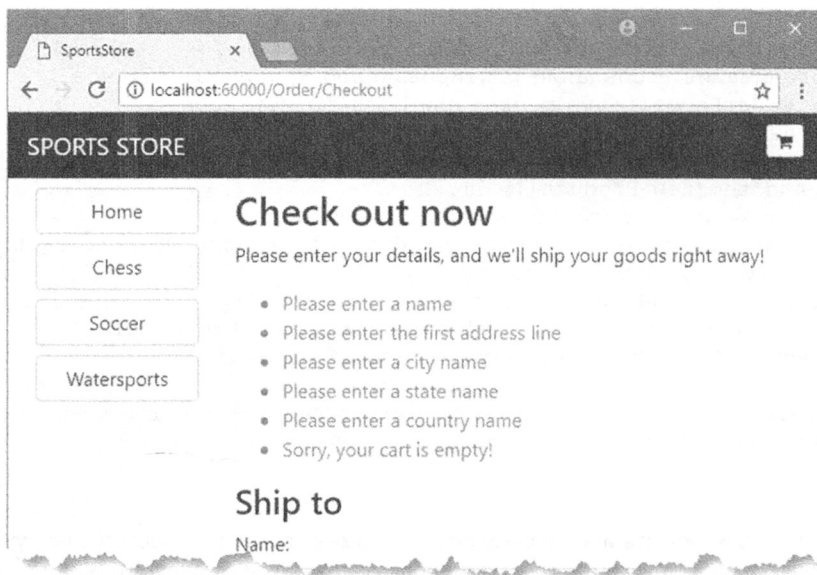


Рис. 10.5. Отображение сообщений об ошибках проверки достоверности

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).

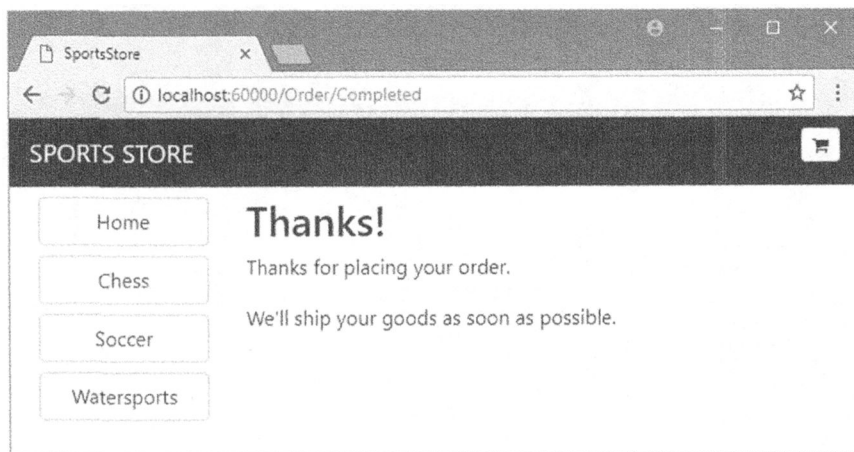


Рис. 10.6. Итоговая страница заверщенного заказа

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). Щелчком на кнопке `Ship` база данных обновится, а список ожидающих заказов будет пуст.

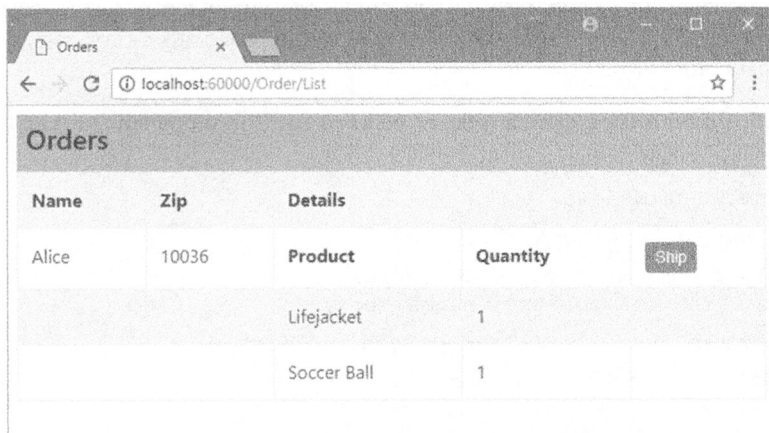


Рис. 11.1. Управление заказами

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).

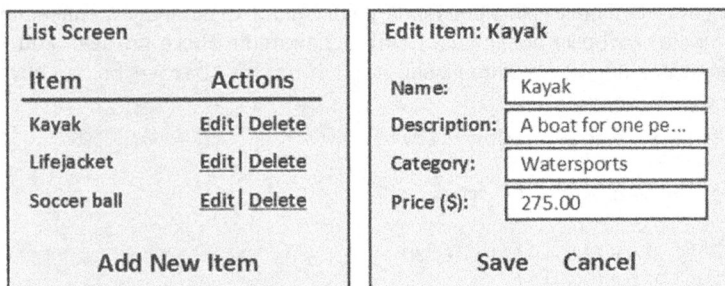


Рис. 11.2. Эскиз пользовательского интерфейса CRUD для каталога товаров

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 IActionResult 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>(ActionResult 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).

ID	Name	Price	Actions
1	Kayak	\$275.00	Edit Delete
2	Lifejacket	\$48.95	Edit Delete
3	Soccer Ball	\$19.50	Edit Delete
4	Corner Flags	\$34.95	Edit Delete
5	Stadium	\$79,500.00	Edit Delete
6	Thinking Cap	\$16.00	Edit Delete
7	Unsteady Chair	\$29.95	Edit Delete
8	Human Chess Board	\$75.00	Edit Delete
9	Bling-Bling King	\$1,200.00	Edit Delete

[Add Product](#)

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 обрабатывает 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 IActionResult Index() => View(repository.Products);

        public IActionResult 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.

The screenshot shows a web browser window with the title 'Edit Product'. The address bar shows 'localhost:60000/Admin/Edit?productId=3'. The form has a header 'Edit Product' and four input fields: 'Name' with the value 'Soccer Ball', 'Description' with the value 'FIFA-approved size and weight', 'Category' with the value 'Soccer', and 'Price' with the value '19.50'. At the bottom of the form are two buttons: 'Save' and 'Cancel'.

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 `IProductRepository.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 `AdminController` 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 IActionResult Index() => View(repository.Products);

        public IActionResult 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 перенаправляется 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
ActionResult 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
    ActionResult 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.

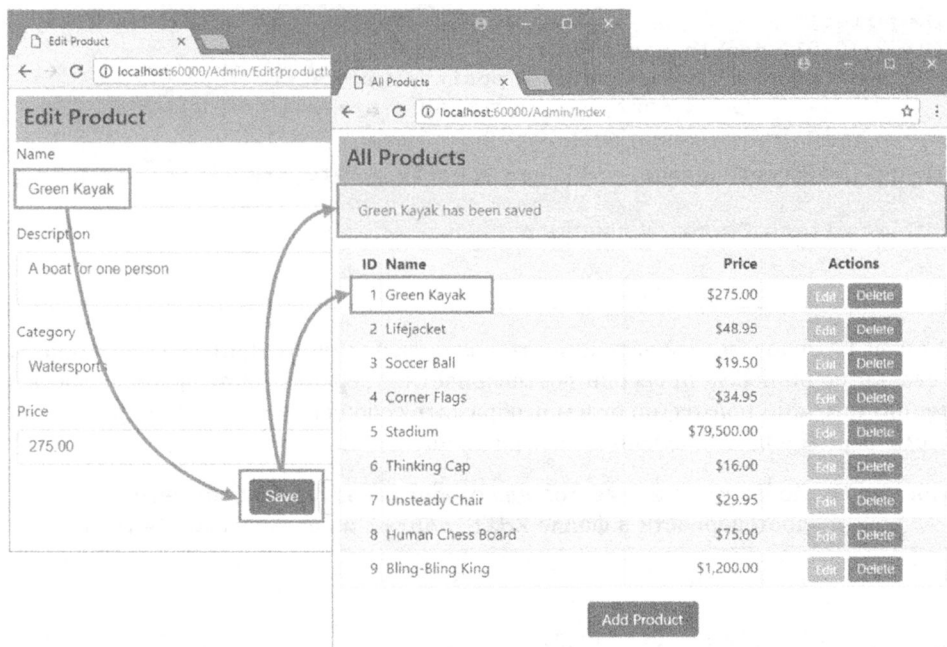


Рис. 11.5. Редактирование сведений о товаре и отображение сообщения из TempData

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.

The screenshot shows a web browser window with the address bar displaying 'localhost:60000/Admin/Edit'. The page title is 'Edit Product'. The form contains the following fields and messages:

- Name:** Placeholder 'Please enter a product name'. The input field has a red border.
- Description:** Placeholder 'Please enter a description'. The input field has a red border.
- Category:** Placeholder 'Please specify a category'. The input field has a red border.
- Price:** Placeholder 'Please enter a positive price'. The input field contains the value '-200' and has a red border.

At the bottom of the form are two buttons: 'Save' and 'Cancel'.

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 `AdminController` 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).

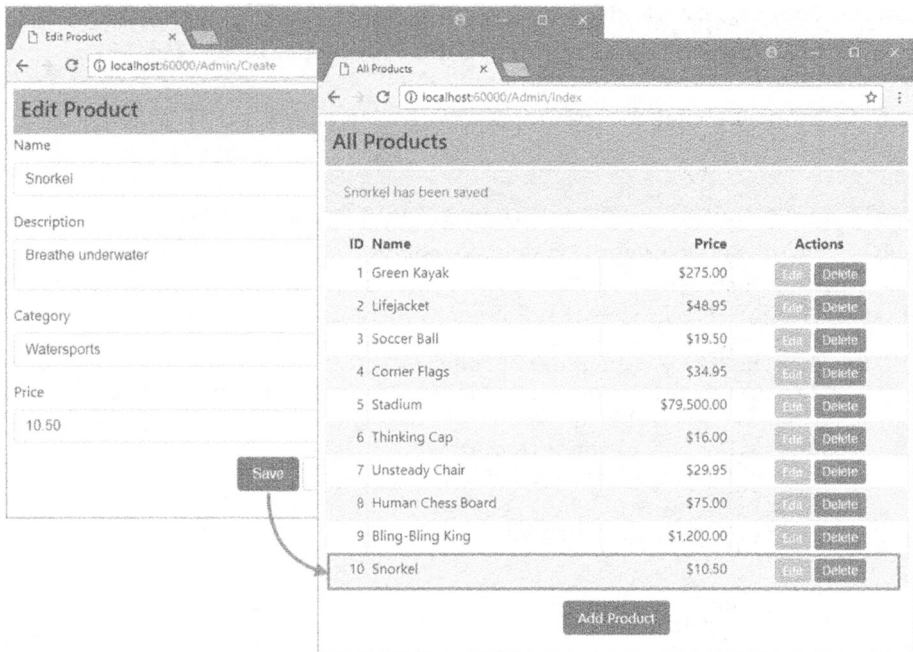


Рис. 11.7. Добавление нового товара в каталог

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 IActionResult Index() => View(repository.Products);

        public IActionResult 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 вызывать 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.

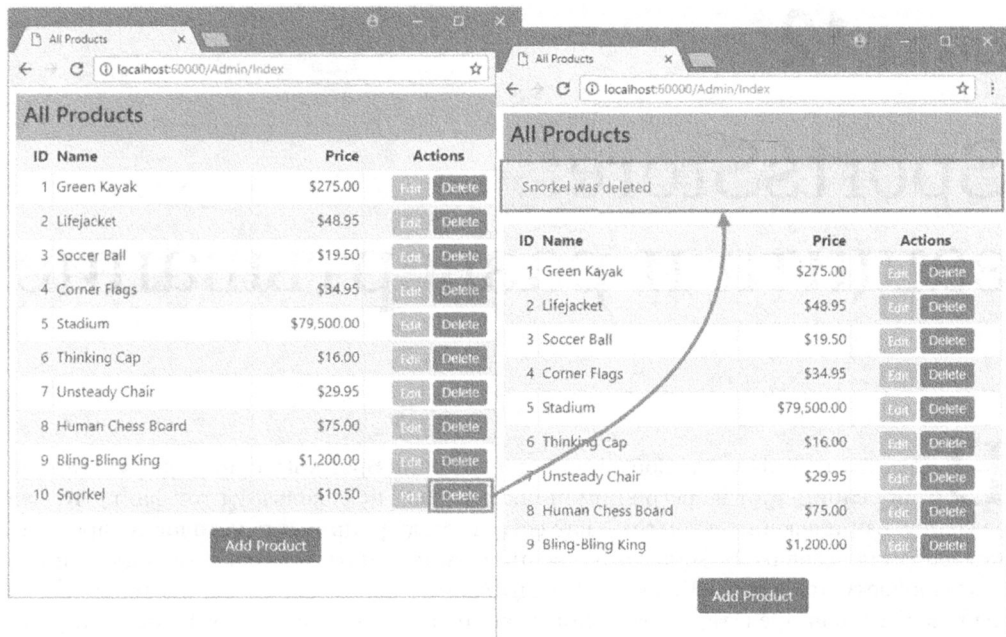


Рис. 11.8. Удаление товара из каталога

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.

ГЛАВА 12

SportsStore: защита и развертывание

В предыдущей главе мы добавили в приложение SportsStore поддержку администрирования, и от вашего внимания, вероятно, не ускользнул тот факт, что если развернуть приложение в том виде как есть, то модифицировать каталог товаров сможет любой пользователь. Для этого ему лишь нужно знать, что средства администрирования доступны через URL вида /Admin/Index и /Order/List. В настоящей главе мы покажем, как предотвратить использование административных функций случайными посетителями, защитив их паролем. После добавления возможности защиты мы объясним, каким образом подготовить и развернуть приложение SportsStore в производственной среде.

Защита средств администрирования

Аутентификация и авторизация предоставляются системой ASP.NET Core Identity, которая аккуратно интегрируется как в платформу ASP.NET Core, так и в приложения MVC. В последующих разделах мы создадим базовую настройку защиты, которая позволит одному пользователю по имени Admin проходить аутентификацию и получать доступ к административным функциям в приложении. Система ASP.NET Core Identity предлагает множество других средств для аутентификации пользователей, а также авторизации доступа к функциям и данным приложения. Более подробные сведения вы найдете в главах 28–30, где будет показано, как создавать и управлять пользовательскими учетными записями, каким образом применять роли и политики и как поддерживать аутентификацию от третьих сторон вроде Microsoft, Google, Facebook и Twitter. Однако цель текущей главы — создать лишь столько функциональности, сколько достаточно для предотвращения доступа пользователей к чувствительным частям приложения SportsStore, что будет содействовать пониманию того, каким образом аутентификация и авторизация вписываются в приложение MVC.

Создание базы данных Identity

Система ASP.NET Core Identity чрезвычайно конфигурируема и расширяема, поддерживая многочисленные варианты хранения данных о пользователях. Мы собираемся использовать наиболее распространенный вариант, который предусматривает хранение данных с применением Microsoft SQL Server и доступ к ним с помощью Entity Framework Core.

Создание класса контекста

Нам необходимо создать файл контекста базы данных, который будет действовать в качестве шлюза между базой данных и объектами моделей базы данных Identity, предоставляющими к ней доступ. Добавим в папку Models файл класса по имени `AppIdentityDbContext.cs` с определением, приведенным в листинге 12.1.

На заметку! Вы могли привыкнуть к тому, что для получения дополнительных функциональных средств вроде защиты к проекту необходимо добавлять пакеты. Но с выпуском ASP.NET Core 2 пакеты NuGet, требующиеся для системы Identity, уже включены в проект через мета-пакет, который был добавлен в файл `SportsStore.csproj` как часть шаблона проекта.

Листинг 12.1. Содержимое файла `AppIdentityDbContext.cs` из папки Models

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace SportsStore.Models {
    public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {
        public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options)
            : base(options) { }
    }
}
```

Класс `AppIdentityDbContext` является производным от класса `IdentityDbContext`, который предлагает связанные с Identity средства для Entity Framework Core. В параметре типа используется `IdentityUser` — встроенный класс, применяемый для представления пользователей. В главе 28 будет продемонстрировано использование специального класса, который можно расширять с целью добавления дополнительной информации о пользователях приложения.

Определение строки подключения

Далее определяется строка подключения, предназначенная для базы данных. В листинге 12.2 показано добавление в файл `appsettings.json` проекта SportsStore, которое следует тому же самому формату, что и строка подключения, определенная для базы данных товаров в главе 8.

Листинг 12.2. Определение строки подключения в файле `appsettings.json` из папки SportsStore

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
        "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    },
    "SportStoreIdentity": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}
```

Вспомните, что строка подключения должна определяться в файле `appsettings.json` как единственная неразрывная строка кода, а в листинге она выглядит так из-за ограниченной ширины печатной страницы. Добавленная разметка определяет строку подключения по имени `SportsStoreIdentity`, в которой указывается база данных `LocalDB` под названием `Identity`.

Конфигурирование приложения

Подобно другим средствам ASP.NET Core система Identity конфигурируется в классе `Start`. В листинге 12.3 приведены добавления для настройки Identity в проекте `SportsStore` с применением ранее определенного класса контекста и строки подключения.

Листинг 12.3. Конфигурирование средства Identity 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;
using Microsoft.AspNetCore.Identity;

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.AddDbContext<AppIdentityDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreIdentity:ConnectionString"]));
            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<AppIdentityDbContext>()
                .AddDefaultTokenProviders();

            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();
        }
    }
}
```

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseMvc(routes => {
        // ...для краткости маршруты не показаны...
    });
    SeedData.EnsurePopulated(app);
}
}
}

```

В методе `ConfigureServices()` конфигурация Entity Framework Core расширена для регистрации класса контекста и с помощью метода `AddIdentity()` устанавливает службы Identity, используя встроенные классы для представления пользователей и ролей. Внутри метода `Configure()` вызывается метод `UseAuthentication()` для установки компонентов, которые будут перехватывать запросы и ответы для внедрения политики безопасности.

Создание и применение миграции базы данных

Основная конфигурация на месте, и самое время воспользоваться средством миграций Entity Framework Core для определения схемы и ее применения к базе данных. Откроем окно командной строки или PowerShell и введем показанную ниже команду в папке проекта SportsStore, чтобы создать новую миграцию для базы данных Identity:

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

Существенное отличие от предшествующих команд базы данных связано с тем, что здесь используется аргумент `--context` для указания имени класса контекста, ассоциированного с базой данных, с которой необходимо работать — `AppIdentityDbContext`. При наличии множества баз данных в приложении важно гарантировать работу с правильным классом контекста.

После того как инфраструктура Entity Framework Core сгенерировала начальную миграцию, выполним следующую команду, чтобы создать базу данных и запустить команды миграции:

```
dotnet ef database update --context AppIdentityDbContext
```

Результатом будет новая база данных LocalDB по имени Identity, которую можно просмотреть с применением проводника объектов SQL Server (SQL Server Object Explorer) среды Visual Studio.

Определение начальных данных

Мы планируем явно создать пользователя Admin, наполняя базу данных начальными данными при запуске приложения. Добавим в папку Models файл класса по имени `IdentitySeedData.cs` и определим в нем статический класс (листинг 12.4).

Листинг 12.4. Содержимое файла IdentitySeedData.cs из папки Models

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {
    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123$";

        public static async void EnsurePopulated(IApplicationBuilder app) {
            UserManager<IdentityUser> userManager = app.ApplicationServices
                .GetRequiredService<UserManager<IdentityUser>>();

            IdentityUser user = await userManager.FindByIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                await userManager.CreateAsync(user, adminPassword);
            }
        }
    }
}
```

В коде используется класс `UserManager<T>`, который система ASP.NET Core Identity предоставляет в виде службы для управления пользователями, как обсуждается в главе 28. В базе данных производится поиск учетной записи пользователя Admin, которая в случае ее отсутствия создается (с паролем Secret123\$). Не изменяйте жестко закодированный пароль в этом примере, поскольку система Identity имеет политику проверки достоверности, которая требует, чтобы пароли содержали цифры и диапазон символов. Способ изменения настроек, относящихся к проверке достоверности, описан в главе 28.

Внимание! Жесткое кодирование деталей учетной записи администратора часто требуется для того, чтобы можно было войти в приложение после его развертывания и начать администрирование. Поступая так, вы должны помнить о необходимости изменения пароля для учетной записи, которую создали. В главе 28 приведены детали того, как изменять пароли, используя Identity.

Чтобы обеспечить начальное заполнение базы данных Identity во время запуска приложения, добавим в метод `Configure()` класса `Startup` оператор, как показано в листинге 12.5.

Листинг 12.5. Начальное заполнение базы данных Identity в файле Startup.cs из папки SportsStore

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
}
```

```

app.UseAuthentication();
app.UseMvc(routes => {
    // ...для краткости маршруты не показаны...
});
SeedData.EnsurePopulated(app);
IdentitySeedData.EnsurePopulated(app);
}
...

```

Применение базовой политики авторизации

Теперь, когда система ASP.NET Core Identity установлена и сконфигурирована, можно применить политику авторизации к тем частям приложения, которые необходимо защитить. Мы собираемся использовать самую базовую политику авторизации, которая предусматривает разрешение доступа любому пользователю, прошедшему аутентификацию. Хотя она может оказаться полезной политикой также и в реальном приложении, существуют возможности для создания более детализированных элементов управления авторизацией (как описано в главах 28–30), но из-за того, что в приложении SportsStore имеется только один пользователь, вполне достаточно провести различие между анонимными и аутентифицированными запросами.

Атрибут `Authorize` применяется для ограничения доступа к методам действий, и в листинге 12.6 видно, что этот атрибут используется для защиты доступа к административным действиям в контроллере `Order`.

Листинг 12.6. Ограничение доступа в файле `OrderController.cs` из папки `Controllers`

```

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

namespace SportsStore.Controllers {
    public class OrderController : Controller {
        private IOrderRepository repository;
        private Cart cart;

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

        [Authorize]
        public ViewResult List() =>
            View(repository.Orders.Where(o => !o.Shipped));

        [HttpPost]
        [Authorize]
        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 IActionResult Checkout() => View(new Order());

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

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

```

Мы не хотим препятствовать доступу пользователей, не прошедших аутентификацию, к остальным методам действий в контроллере `Order`, так что атрибут `Authorize` применен только к методам `List()` и `MarkShipped()`. Нам нужно защитить все методы действий, определяемые контроллером `Admin`, чего можно достичь за счет применения атрибута `Authorize` к самому классу контроллера, что приведет к применению политики авторизации ко всем содержащимся в нем методам действий (листинг 12.7).

Листинг 12.7. Ограничение доступа в файле `AdminController.cs` из папки `Controllers`

```

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

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

```

Создание контроллера Account и представлений

Когда пользователь, не прошедший аутентификацию, посылает запрос, который требует авторизации, он перенаправляется на URL вида /Account/Login, где приложение может предложить пользователю ввести свои учетные данные. В качестве подготовки создадим модель представления для учетных данных пользователя, добавив в папку Models/ViewModels файл класса по имени LoginModel.cs с определением, показанным в листинге 12.8.

Листинг 12.8. Содержимое файла LoginModel.cs из папки Models/ViewModels

```

using System.ComponentModel.DataAnnotations;
namespace SportsStore.Models.ViewModels {
    public class LoginModel {
        [Required]
        public string Name { get; set; }

        [Required]
        [UIHint("password")]
        public string Password { get; set; }

        public string returnUrl { get; set; } = "/";
    }
}

```

Свойства `Name` и `Password` декорированы атрибутом `Required`, который использует проверку достоверности модели для обеспечения того, что значения были предоставлены. Свойство `Password` декорировано атрибутом `UIHint`, поэтому в случае применения атрибута `asp-for` внутри элемента `input` представления `Razor`, предназначенного для входа, вспомогательная функция дескриптора установит атрибут `type` в `password`; таким образом, вводимый пользователем текст не будет виден на экране. Использование атрибута `UIHint` описано в главе 24.

Далее добавим в папку `Controllers` файл класса по имени `AccountController.cs` с определением контроллера, приведенным в листинге 12.9. Данный контроллер будет отвечать на запросы к URL вида `/Account/Login`.

Листинг 12.9. Содержимое файла `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {
    [Authorize]
    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManager;

        public AccountController(UserManager<IdentityUser> userMgr,
            SignInManager<IdentityUser> signInMgr) {
            userManager = userMgr;
            signInManager = signInMgr;
        }

        [AllowAnonymous]
        public ViewResult Login(string returnUrl) {
            return View(new LoginModel {
                ReturnUrl = returnUrl
            });
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginModel loginModel) {
            if (ModelState.IsValid) {
                IdentityUser user =
                    await userManager.FindByNameAsync(loginModel.Name);
                if (user != null) {
                    await signInManager.SignOutAsync();
                    if ((await signInManager.PasswordSignInAsync(user,
                        loginModel.Password, false, false)).Succeeded) {
                        return Redirect(loginModel?.ReturnUrl ?? "/Admin/Index");
                    }
                }
            }
        }
    }
}
```

```

ModelState.AddModelError("", "Invalid name or password");
// Неправильное имя или пароль
return View(loginModel);
}
public async Task<RedirectResult> Logout(string returnUrl = "/") {
    await signInManager.SignOutAsync();
    return Redirect(returnUrl);
}
}
}

```

Когда пользователь перенаправляется на URL вида /Account/Login, версия GET метода действия Login() визуализирует стандартное представление для страницы и создает объект модели представления, включающий URL, на который браузер должен быть перенаправлен, если запрос на аутентификацию завершился успешно.

Учетные данные аутентификации отправляются версии POST метода действия Login(), которая применяет службы UserManager<IdentityUser> и SignInManager<IdentityUser>, полученные через конструктор класса контроллера, для аутентификации пользователя и его входа в систему. Работа упомянутых классов объясняется в главах 28–30, а пока достаточно знать, что в случае отказа в аутентификации создается ошибка проверки достоверности модели и визуализируется стандартное представление. Если же аутентификация прошла успешно, тогда пользователь перенаправляется на URL, к которому он хотел получить доступ перед тем, как ему было предложено ввести свои учетные данные.

Внимание! В целом использование проверки достоверности данных на стороне клиента является хорошей практикой. Она освобождает от определенной работы сервер и обеспечивает пользователям немедленный отклик о предоставленных ими данных. Тем не менее, не поддавайтесь искушению выполнять на стороне клиента аутентификацию, поскольку обычно это предусматривает передачу клиенту допустимых учетных данных, которые будут применяться при проверке вводимых имени пользователя и пароля, или, по меньшей мере, наличие доверия сообщению клиента о том, что аутентификация завершилась успешно. Аутентификация должна всегда выполняться на сервере.

Чтобы снабдить метод Login() представлением для визуализации, создадим папку Views/Account и поместим в нее файл представления Razor по имени Login.cshtml с содержимым, показанным в листинге 12.10.

Листинг 12.10. Содержимое файла Login.cshtml из папки Views/Account

```

@model LoginModel
@{
    ViewBag.Title = "Log In";
    Layout = "_AdminLayout";
}
<div class="text-danger" asp-validation-summary="All"></div>
<form asp-action="Login" asp-controller="Account" method="post">
    <input type="hidden" asp-for="ReturnUrl" />
    <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="Password"></label>
  <div><span asp-validation-for="Password" class="text-danger"></span>
  </div>
  <input asp-for="Password" class="form-control" />
</div>
<button class="btn btn-primary" type="submit">Log In</button>
</form>

```

Финальный шаг связан с изменением разделяемой компоновки для администрирования, чтобы добавить кнопку Log Out (Выход), которая позволит текущему пользователю выходить из приложения за счет отправки запроса действию Logout (листинг 12.11). Это удобное средство, облегчающее тестирование приложения, без которого пришлось бы очищать cookie-наборы браузера, чтобы возвращаться в состояние, когда аутентификация еще не прошла.

Листинг 12.11. Добавление кнопки Log Out 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 row">
    <div class="col">
      <h4>@ViewBag.Title</h4>
    </div>
    <div class="col-2">
      <a class="btn btn-sm btn-primary"
        asp-action="Logout" asp-controller="Account">Log Out</a>
    </div>
  </div>
  @if (TempData["message"] != null) {
    <div class="alert alert-success mt-1">@TempData["message"]</div>
  }
  @RenderBody()
</body>
</html>

```

Тестирование политики безопасности

Теперь можно протестировать политику безопасности, запустив приложение и запросив URL вида `/Admin/Index`. Поскольку в настоящий момент мы еще не прошли аутентификацию и пытаемся обратиться к действию, которое требует авторизации, браузер будет перенаправлен на URL вида `/Account/Login`. Введем **Admin** и **Secret123\$** в качестве имени и пароля и отправим форму. Контроллер `Account` сравнит предоставленные учетные данные с начальными данными, добавленными в базу данных `Identity`, и (при условии ввода правильных сведений) аутентифицирует нас, после чего перенаправит обратно на `/Account/Login`, куда теперь имеется доступ. Процесс проиллюстрирован на рис. 12.1.

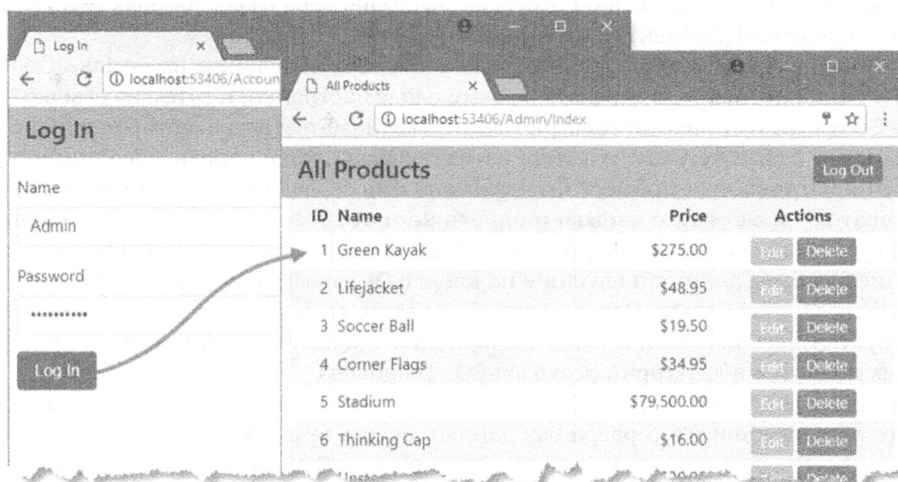


Рис. 12.1. Процесс аутентификации/авторизации для административных функций

Развертывание приложения

Все средства и функциональность приложения `SportsStore` на месте, так что наступил момент для его подготовки и развертывания в производственной среде. В отношении приложений `ASP.NET Core MVC` доступно множество вариантов размещения, и в настоящей главе используется один из них — платформа `Microsoft Azure`. Она выбрана из-за того, что поступает от `Microsoft` и предлагает бесплатные учетные записи, т.е. вы можете полностью проработать пример `SportsStore`, даже если не хотите применять `Azure` в собственных проектах.

На заметку! В данном разделе понадобится учетная запись `Azure`. Если у вас ее пока нет, тогда можете создать бесплатную учетную запись на <https://azure.microsoft.com/>.

Создание баз данных

Начальной задачей является создание баз данных, которые приложение `SportsStore` будет использовать в производственной среде. Такую задачу можно выполнять как часть процесса разработки в `Visual Studio`, но трудность ситуации в том, что нужно

знать строки подключений для баз данных до развертывания, а это относится к процессу, который создает базы данных.

Внимание! Портал Azure часто меняется по мере того, как в Microsoft добавляют новые средства и пересматривают существующие. Инструкции, приводимые в настоящем разделе, были точными на время его написания, но к моменту выхода книги необходимые шаги могут слегка измениться. Базовый подход должен остаться таким же, но имена полей данных и точный порядок шагов может потребовать определенного экспериментирования, чтобы добиться правильных результатов.

Самый простой подход предусматривает вход на портал <https://portal.azure.com> с применением своей учетной записи Azure и создание баз данных вручную. После входа выберем категорию ресурсов SQL Databases (Базы данных SQL) и щелкнем на кнопке Add (Добавить), чтобы создать новую базу данных.

Для первой базы данных укажем имя **products**. Щелкнем на ссылке Configure Required Settings (Конфигурировать обязательные настройки) и затем на ссылке Create a New Server (Создать новый сервер). Введем имя нового сервера, которое должно быть уникальным в рамках Azure, и выберем имя пользователя и пароль для администратора. В рассматриваемом примере было указано имя сервера **sportsstorecore2db**, имя пользователя **sportsstoreadmin** и пароль **Secret123\$**. Вам понадобится использовать другое имя сервера и сформировать более надежный пароль. Выберем местоположение для базы данных и щелкнем на кнопке OK, чтобы закрыть экран параметров, и далее на кнопке Create (Создать), чтобы создать саму базу данных. Порталу Azure потребуется несколько минут для выполнения процесса создания, после чего база данных появится в категории ресурсов SQL Databases.

Создадим еще один сервер баз данных SQL, указав на этот раз имя **identity**. Вместо создания нового сервера баз данных можно применить тот, который был создан ранее. Результатом окажутся две базы данных SQL Server, размещаемые Azure, детали которых приведены в табл. 12.1. У вас будут другие имена серверов баз данных и наверняка более надежные пароли.

Таблица 12.1. Базы данных Azure для приложения SportsStore

Имя базы данных	Имя сервера	Имя пользователя-администратора	Пароль
products	sportsstorecore2db	sportsstoreadmin	Secret123\$
identity	sportsstorecore2db	sportsstoreadmin	Secret123\$

Открытие доступа в брандмауэре для конфигурирования

Далее необходимо создать схемы баз данных, и сделать это проще всего, открыв доступ в брандмауэре Azure, чтобы можно было запускать команды Entity Framework Core из машины разработки.

Выберем одну из двух баз данных в категории ресурсов SQL Databases, щелкнем на кнопке Tools (Сервис) и затем щелкнем на ссылке Open in Visual Studio (Открыть в Visual Studio). Теперь щелкнем на ссылке Configure Your Firewall (Конфигурировать брандмауэр), щелкнем на кнопке Add Client IP (Добавить IP-адрес клиента) и щелкнем на кнопке Save (Сохранить). В результате текущий IP-адрес получит возможность достигать сервера баз данных и выполнять команды конфигурирования. (Проинспектировать схему базы данных можно, щелкнув на кнопке Open In Visual Studio, чтобы открыть

Visual Studio и воспользоваться окном SQL Server Object Explorer для исследования базы данных.)

Получение строк подключений

Вскоре понадобятся строки подключений для новых баз данных. Портал Azure предоставляет такую информацию по щелчку на базе данных в категории ресурсов SQL Databases через ссылку Show Database Connection Strings (Показать строки подключений для баз данных). Строки подключений предлагаются для разных платформ разработки; приложениям .NET требуются строки ADO.NET. Вот строка подключения, которую портал Azure предоставляет для базы данных products:

```
Server=tcp:sportsstorecore2db.database.windows.net,1433;
Initial Catalog=products;
Persist Security Info=False;
User ID={ваше_имя_пользователя};Password={ваш_пароль};
MultipleActiveResultSets=True;Encrypt=True;
TrustServerCertificate=False;Connection Timeout=30;
```

В зависимости от того, как портал Azure подготовил базу данных, вы будете видеть разные параметры конфигурации. Обратите внимание на выделенные полужирным заполнители для имени пользователя и пароля, которые должны быть изменены, когда вы применяете строку подключения при конфигурировании приложения.

Подготовка приложения

Перед тем, как приложение можно будет развернуть, предстоит выполнить определенные подготовительные шаги, чтобы привести его в готовность к производственной среде. В последующих разделах будет изменен способ отображения сообщений об ошибках и настроены строки подключения для производственных баз данных.

Создание контроллера и представления для отображения сообщений об ошибках

В текущий момент приложение сконфигурировано на использование страниц ошибок, дружественных к разработчику, которые предоставляют полезную информацию при возникновении проблемы. Конечные пользователи не должны видеть такую информацию, поэтому добавим в папку Controllers файл класса по имени `ErrorController.cs` с определением простого контроллера, показанного в листинге 12.12.

Листинг 12.12. Содержимое файла `ErrorController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {
    public class ErrorController : Controller {
        public IActionResult Error() => View();
    }
}
```

В контроллере определено действие `Error`, которое визуализирует стандартное представление. Чтобы снабдить контроллер представлением, создадим папку `Views/Error` и добавим в нее файл представления Razor по имени `Error.cshtml` с разметкой, приведенной в листинге 12.13.

Листинг 12.13. Содержимое файла Error.cshtml из папки Views/Error

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <title>Error</title>
</head>
<body>
    <h2 class="text-danger">Error.</h2>
    <h3 class="text-danger">An error occurred while processing your request.</h3>
</body>
</html>
```

Страница ошибки подобного рода является последним средством, а потому ее лучше сохранить как можно более простой и не полагаться на разделяемые компоновки, компоненты представлений или другие расширенные возможности. В данном случае мы отключаем разделяемые компоновки и определяем простой HTML-документ с сообщением о возникновении ошибки, не предоставляя никакой информации о том, что произошло.

Определение настроек производственных баз данных

Следующим шагом будет создание файла, который снабдит приложение строками подключения к его базам данных в производственной среде. Добавим в проект SportsStore новый файл по имени appsettings.production.json с применением шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и поместим в него содержимое, показанное в листинге 12.14.

Совет. В списке файлов окна Solution Explorer файл appsettings.production.json находится внутри узла appsettings.json, который понадобится раскрыть, если позже вы захотите отредактировать данный файл.

Листинг 12.14. Содержимое файла appsettings.production.json из папки SportsStore

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
        "Server=tcp:sportsstorecore2db.database.windows.net,1433;
        Initial Catalog=products;Persist Security Info=False;
        User ID={ваше_имя_пользователя};
        Password={ваш_пароль};MultipleActiveResultSets=True;Encrypt=True;
        TrustServerCertificate=False;Connection Timeout=30;"
    },
    "SportStoreIdentity": {
      "ConnectionString":
        "Server=tcp:sportsstorecore2db.database.windows.net,1433;
```

```

Initial Catalog=identity;Persist Security Info=False;
User ID={ваше_имя_пользователя};
Password={ваш_пароль};MultipleActiveResultSets=True;Encrypt=True;
TrustServerCertificate=False;Connection Timeout=30;"
    }
}
}

```

Файл неудобен для чтения, т.к. строки подключений разрывать нельзя. Содержимое данного файла дублирует раздел строк подключений файла `appsettings.json`, но здесь используются строки подключений Azure. (Не забудьте заменить заполнители для имени пользователя и пароля.) Кроме того, параметр `MultipleActiveResultSets` был установлен в `True`, что делает возможными множество параллельных запросов и устраняет условия для возникновения распространенной ошибки, которые появляются при выполнении сложных запросов LINQ в отношении данных приложения.

На заметку! При вставке в строки подключения своего имени пользователя и пароля удалите символы фигурных скобок, чтобы в итоге получилось `Password=MyPassword`, но не `Password={MyPassword}`.

Конфигурирование приложения

Теперь можно изменить код класса `Startup`, чтобы в случае нахождения в производственной среде приложение вело себя по-другому. Изменения представлены в листинге 12.15.

Листинг 12.15. Конфигурирование приложения 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;
using Microsoft.AspNetCore.Identity;

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.AddDbContext<AppIdentityDbContext>(options =>
    options.UseSqlServer(
        Configuration["Data:SportStoreIdentity:ConnectionString"]));
services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AppIdentityDbContext>()
    .AddDefaultTokenProviders();

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();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
    } else {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseMvc(routes => {
        routes.MapRoute(name: "Error", template: "Error",
            defaults: new { controller = "Error", action = "Error" });
        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);
    // IdentitySeedData.EnsurePopulated(app);
}
}

```

Интерфейс `IHostingEnvironment` используется для предоставления информации о среде, в которой функционирует приложение, такой как среда разработки или производственная среда. В случае производственной среды ASP.NET Core будет загружать содержимое файла `appsettings.production.json`, чтобы переопределить настройки в файле `appsettings.json`, а это значит, что Entity Framework Core будет подключаться к базам данных Azure вместо LocalDB. Доступно множество параметров для настройки конфигурации приложения в различных средах, которые будут рассматриваться в главе 14.

Также были закомментированы операторы, которые производят начальное заполнение баз данных, что объясняется в разделе "Управление начальным заполнением баз данных" далее в главе.

Применение миграций к базам данных

Чтобы настроить базы данных со схемами, требующимися приложению, откроем окно командной строки или PowerShell и перейдем в папку проекта SportsStore. Установка среды так, что инструмент командной строки `dotnet` будет использовать строки подключения для Azure, требует установки переменной среды. В случае применения окна PowerShell установка переменной среды выполняется с помощью следующей команды:

```
$env:ASPNETCORE_ENVIRONMENT="Production"
```

Если же используется окно командной строки, то переменная среды устанавливается посредством такой команды:

```
set ASPNETCORE_ENVIRONMENT=Production
```

Для применения миграций в проекте к базам данных Azure нужно запустить показанные далее команды, находясь в папке проекта SportsStore:

```
dotnet ef database update --context ApplicationDbContext
dotnet ef database update --context AppIdentityDbContext
```

Переменная среды указывает размещающую среду, которая применяется для получения строк подключения к базам данных. Если приведенные команды не работают, тогда понадобится проверить, сконфигурирован ли брандмауэр Azure для разрешения доступа к машине разработки, как было описано ранее в главе, а также корректно ли скопированы и модифицированы строки подключения.

Управление начальным заполнением баз данных

В листинге 12.15 операторы, выполняющие начальное заполнение баз данных в классе `Startup`, закомментированы. Дело в том, что команды Entity Framework Core, используемые в предыдущем разделе для применения миграций к базе данных, полагаются на службы, устанавливаемые классом `Startup`. При наличии упомянутых операторов код начального заполнения баз данных вызывался бы перед применением миграций, приводя к ошибке и препятствуя нормальной работе миграций. Когда базы данных установлены, проблема не возникает. В случае базы данных `products` причина в том, что метод `SeedData.EnsurePopulated()` применяет миграции перед заполнением начальными данными, и в том, что начальные данные `Identity` не добавляются в приложение до тех пор, пока к базе данных не будет применена миграция.

В отношении производственной среды мы хотим принять другой подход к заполнению начальными данными. Для пользовательских учетных записей мы собираемся поместить в базу данных учетную запись администратора, когда производится попытка входа. Мы добавим к инструменту администрирования средство для начального заполнения базы данных товаров, чтобы производственная система могла заполняться тестовыми данными или оставаться пустой для поступления реальных данных.

На заметку! Начальное заполнение данными аутентификации в производственной системе должно делаться осторожно, а приложение обязано использовать функциональные средства, описанные в главах 28–30, для изменения пароля сразу же после того, как приложение развернуто.

Начальное заполнение данными о пользователях

Первый шаг в изменении способа заполнения данными о пользователях предусматривает упрощение кода в классе `IdentitySeedData` (листинг 12.16).

Листинг 12.16. Упрощение кода в файле `IdentitySeedData.cs` из папки `Models`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.Tasks;

namespace SportsStore.Models {
    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123$";

        public static async Task EnsurePopulated(UserManager<IdentityUser>
            userManager) {

            IdentityUser user = await userManager.FindByIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                await userManager.CreateAsync(user, adminPassword);
            }
        }
    }
}
```

Вместо получения самой службы `UserManager<IdentityUser>` метод `EnsurePopulated()` получает в качестве аргумента объект, что позволяет интегрировать начальное заполнение базы данных в класс `AccountController` (листинг 12.17).

Листинг 12.17. Начальное заполнение базы данных в файле `AccountController.cs` из папки `Controllers`

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;
```

```

using SportsStore.Models;

namespace SportsStore.Controllers {
    [Authorize]
    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManager;

        public AccountController(UserManager<IdentityUser> userMgr,
            SignInManager<IdentityUser> signInMgr) {
            userManager = userMgr;
            signInManager = signInMgr;
            IdentitySeedData.EnsurePopulated(userMgr).Wait();
        }

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

```

Внесенные изменения гарантируют, что база данных Identity будет заполняться каждый раз, когда создается объект AccountController для обработки HTTP-запроса. Конечно, это далеко от идеала, но не существует хорошего способа начального заполнения базы данных, и такой подход обеспечивает возможность администрирования приложения в производственной среде и среде разработки, хотя за счет ряда дополнительных запросов к базе данных.

Начальное заполнение данными о товарах

Что касается данных о товарах, то мы собираемся снабдить администратора кнопкой, щелчок на которой приведет к начальному заполнению базы данных, когда она пуста. Первым делом изменим код начального заполнения, чтобы задействовать в нем интерфейс, который позволит обращаться к службам, предоставляемым контроллером, а не классом Startup (листинг 12.18). Кроме того, мы закомментировали операторы для автоматического применения любых ожидающих миграций, которые могут привести к утере данных и должны использоваться с особой осторожностью в производственных системах.

Листинг 12.18. Подготовка к ручному начальному заполнению в файле SeedData.cs из папки Models

```

using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using System;

namespace SportsStore.Models {
    public static class SeedData {
        public static void EnsurePopulated(IServiceProvider services) {
            ApplicationDbContext context =
                services.GetRequiredService<ApplicationDbContext>();

            // context.Database.Migrate();
            if (!context.Products.Any()) {
                context.Products.AddRange(

```

```

        // ...для краткости другие операторы не показаны...
    };
    context.SaveChanges();
}
}
}
}
}

```

Следующий шаг заключается в обновлении контроллера Admin с целью добавления метода действия, который инициирует операцию начального заполнения (листинг 12.19).

Листинг 12.19. Начальное заполнение базы данных в файле AdminController.cs из папки Controllers

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using Microsoft.AspNetCore.Authorization;
namespace SportsStore.Controllers {
    [Authorize]
    public class AdminController : Controller {
        private IProductRepository repository;
        public AdminController(IProductRepository repo) {
            repository = repo;
        }
        public ViewResult Index() => View(repository.Products);
        // ...для краткости другие методы не показаны...
        [HttpPost]
        public IActionResult SeedDatabase() {
            SeedData.EnsurePopulated(HttpContext.RequestServices);
            return RedirectToAction(nameof(Index));
        }
    }
}

```

Новое действие декорируется атрибутом HttpPost, так что оно может быть целью запросов POST, и после начального заполнения базы данных перенаправляет браузер на метод действия Index(). Осталось лишь создать кнопку для начального заполнения базы данных, которая будет отображаться, когда база данных пуста (листинг 12.20).

Листинг 12.20. Добавление кнопки для начального заполнения базы данных в файле Index.cshtml из папки Views/Admin

```

@model IEnumerable<Product>
@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}

```

```

@if (Model.Count() == 0) {
    <div class="text-center m-2">
        <form asp-action="SeedDatabase" method="post">
            <button type="submit" class="btn btn-danger">Seed Database</button>
        </form>
    </div>
} else {
    <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>

```

Развертывание приложения

Чтобы развернуть приложение, щелкнем правой кнопкой мыши на элементе проекта SportsStore в окне Solution Explorer (проекта, но не решения) и выберем в контекстном меню пункт Publish (Опубликовать). Среда Visual Studio предложит выбрать метод опубликования (рис. 12.2).

Что делать, если развертывание терпит неудачу?

Самой главной причиной неудачи развертывания являются строки подключения, либо потому, что они были некорректно скопированы из среды Azure, либо из-за того, что при редактировании для вставки имени пользователя и пароля была допущена ошибка.

В случае отказа развертывания начинать поиск проблемы нужно со строк подключения. Если команды `dotnet ef database update` из раздела “Применение миграций к базам данных” выше в главе не приводят к ожидаемым результатам, то развертывание потерпит неудачу. Если команды отработали успешно, но развертывание закончилось неудачей, тогда необходимо проверить, установлена ли переменная среды, поскольку есть вероятность того, что вместо базы данных в облаке была подготовлена локальная база данных.

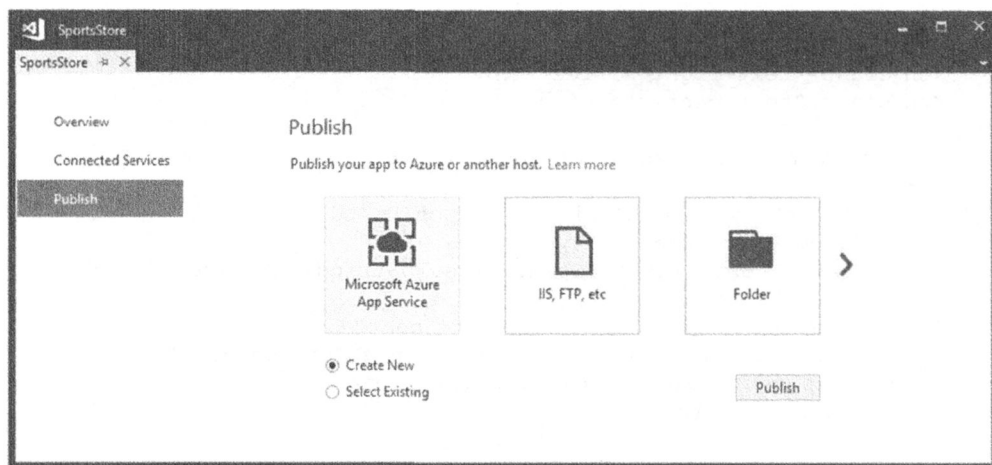


Рис. 12.2. Выбор метода опубликования

Выберем вариант Microsoft Azure App Service (Служба приложений Microsoft Azure) и удостоверимся в выборе переключателя Create New (Создать новую); переключатель Select Existing (Выбрать существующую) используется для обновления существующего развернутого приложения. Будет предложено предоставить детали для развертывания. Начнем со щелчка на кнопке Add an Account (Добавить учетную запись) и ввода учетных данных Azure.

После ввода учетных данных можно выбрать имя для развернутого приложения и ввести информацию относительно службы, которая будет зависеть от типа имеющейся учетной записи Azure, желаемой области для развертывания и требующейся службы развертывания (рис. 12.3).

После того, как служба сконфигурирована, щелчком на кнопке Create (Создать). Как только служба установится, будет выдана сводка по операции опубликования, которая отправит приложение в службу размещения (рис. 12.4).

Щелкнем на кнопке Publish (Опубликовать), чтобы запустить процесс развертывания. Выбрав пункт Web Publish Activity (Активность веб-развертывания) в меню View⇒Other Windows (Вид⇒Другие окна) среды Visual Studio, можно наблюдать за ходом развертывания. Наберитесь терпения, т.к. отправка всех файлов из проекта службе Azure может занять некоторое время. Последующие обновления будут проходить быстрее, потому что передаче подлежат только измененные файлы.

По завершении развертывания среда Visual Studio откроет новое окно браузера для развернутого приложения. Поскольку база данных товаров пуста, отобразится компоновка, показанная на рис. 12.5.

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account

Hosting Services

App Name: SportsStoreCore-2

Subscription: Visual Studio Enterprise

Resource Group: [New...]

App Service Plan: SportsStoreCore2Plan* [New...]

Clicking the Create button will create the following Azure resources

- Explore additional Azure services
- App Service - SportsStoreCore-2
- App Service Plan - SportsStoreCore2Plan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources. [Learn More](#)

Export... Create Cancel

Рис. 12.3. Создание новой службы приложений Azure

SportsStore - SportsStore

SportsStore

Overview

Connected Services

Publish

Publish

Publish your app to Azure or another host. [Learn more](#)

SportsStoreCore-2 - Web Deploy [Publish]

Create new profile

Summary

Site URL	http://sportsstorecore-2.azurewebsites.net	Settings...
Resource Group	core2	Preview...
Configuration	Release	Rename profile...
Username	SSportsStoreCore-2	Delete profile
Password	*****	

Рис. 12.4. Сводка по операции опубликования

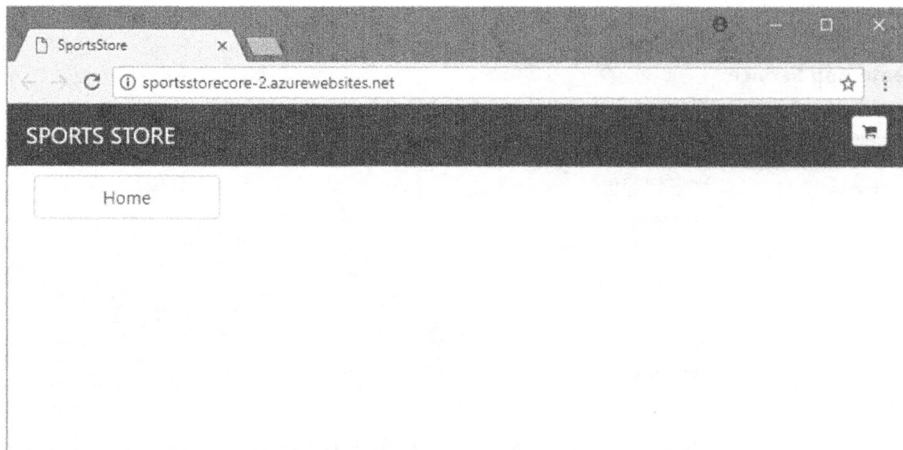


Рис. 12.5. Начальное состояние развернутого приложения

Перейдем на URL вида /Admin/Index и для аутентификации введем имя пользователя **Admin** и пароль **Secret123\$**. Произойдет начальное заполнение базы данных Identity по требованию, что даст возможность войти в административную часть приложения (рис. 12.6).

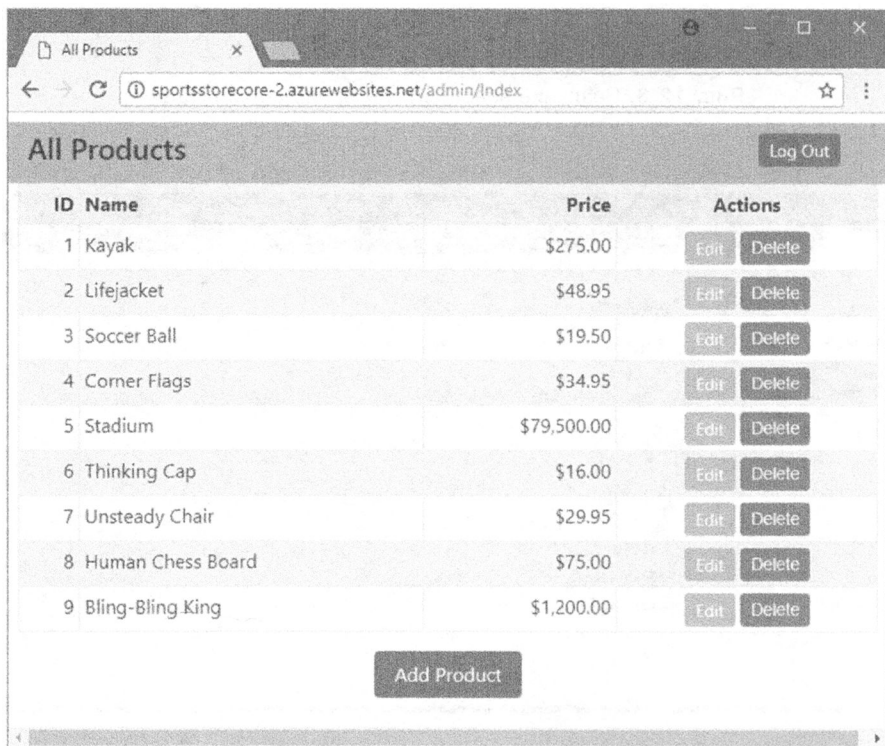


Рис. 12.6. Экран администрирования

Щелчком на кнопке Seed Database (Заполнить базу данных), чтобы заполнить базу данных товаров, что приведет к результату, представленному на рис. 12.7. Затем можно перейти на корневой URL приложения и работать с ним обычным образом.

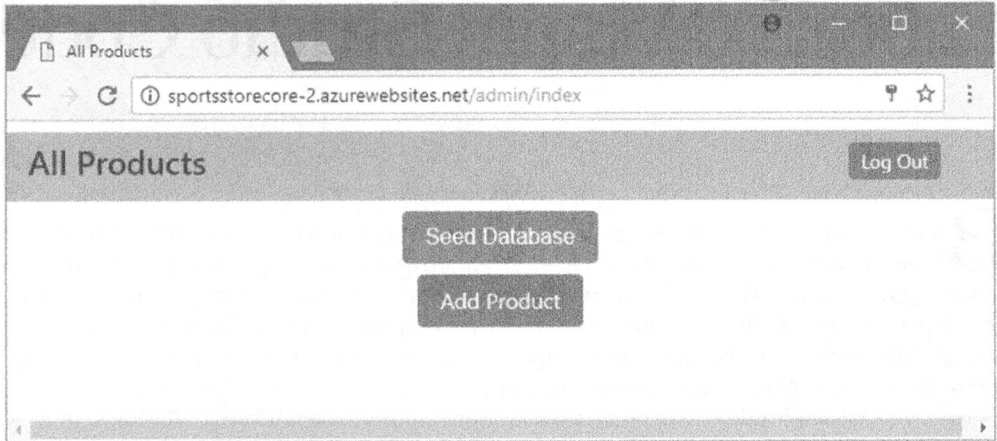


Рис. 12.7. Заполненная база данных

Резюме

В этой и предшествующих главах демонстрировалось применение инфраструктуры ASP.NET Core MVC для создания реалистичного приложения электронной коммерции. В приведенном расширенном примере были проиллюстрированы многие основные средства MVC: контроллеры, методы действий, маршрутизация, представления, метаданные, проверка достоверности, компоновки, аутентификация и т.д. Вы также видели, как пользоваться рядом ключевых технологий, имеющих отношение к MVC, в том числе Entity Framework Core, внедрение зависимостей и модульное тестирование. Результатом стало приложение с ясной и ориентированной на компоненты архитектурой, обеспечивающей разделение обязанностей, и кодовой базой, которую будет легко расширять и сопровождать. Итак, разработка приложения SportsStore завершена. В следующей главе будет показано, как применять Visual Studio Code для создания приложений ASP.NET Core MVC.