ГЛАВА 8

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 онлайновых магазинах. Мы создадим онлайновый каталог товаров, который потребители могут просматривать по категориям и страницам, корзину для покупок, куда пользователи могут добавлять и удалять товары, и форму оплаты, где потребители могут 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 этой 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, то можете смело пропускать эти 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.

На заметку! В Microsoft заявили, что в следующей версии Visual Studio изменят инструментарий, применяемый для создания приложений ASP.NET Core MVC. Проверяйте веб-сайт издательства на предмет обновлений, которые появятся после выпуска новых инструментов.

Начало работы

Если вы планируете писать код приложения SportsStore на своем компьютере во время изучения материала этой части книги, то вам придется установить Visual Studio и удостовериться в том, что установлен вариант LocalDB, который требуется для постоянного хранения данных.

На заметку! Если вы просто хотите работать с проектом, не воссоздавая его, тогда можете загрузить готовый проект SportsStore как часть загружаемого кода примеров для настоящей книги, который доступен на веб-сайте издательства. 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 эту 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. Выберите в меню File (Файл) среды Visual Studio пункт New⇨Project (Создать⇨Проект) и укажите шаблон проекта ASP.NET Core Web Application (.NET Core) (Веб-приложение ASP.NET Core (.NET Core)), как показано на рис. 8.1. В качестве имени проекта введите `SportsStore` и щелкните на кнопке OK.

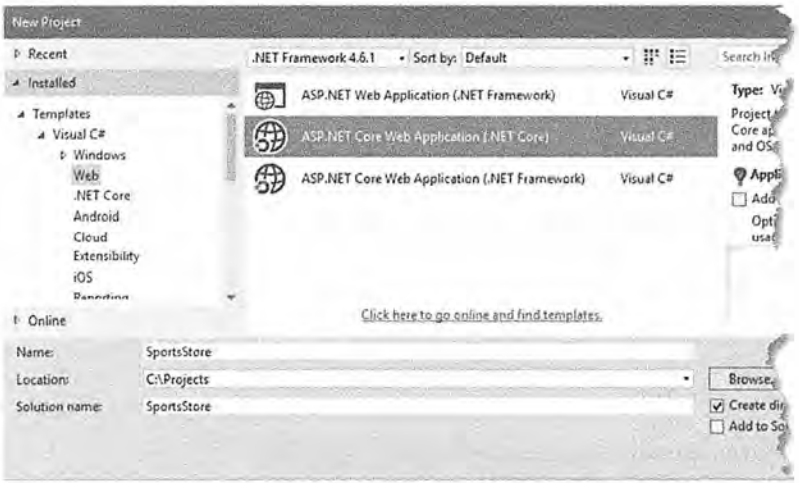

Рис. 8.1. Выбор типа проекта

Выберите шаблон Empty (Пустой), как проиллюстрировано на рис. 8.2, и щелкните на кнопке OK, чтобы создать проект `SportsStore`.

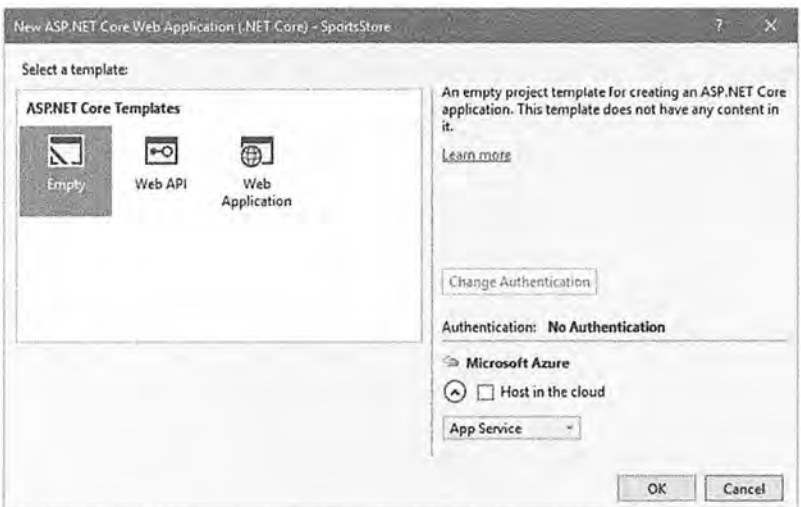

Рис. 8.2. Выбор шаблона проекта

Добавление пакетов NuGet

Шаблон проекта Empty устанавливает базовые средства ASP.NET Core, но требуются дополнительные пакеты, чтобы предоставить функциональность, обязательную для приложений MVC. В листинге 8.1 приведены изменения, внесенные в файл `project.json`, которые добавляют пакеты, необходимые для того, чтобы начать разработку приложения SportsStore.

Листинг 8.1. Добавление пакетов NuGet в файле `project.json`

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.0.0",
      "type": "platform"
    },
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",
    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    },
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0"
  },
  "tools": {
    "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
    "Microsoft.AspNetCore.Server.IISIntegration.Tools":
      "1.0.0-preview2-final"
    },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": ["dotnet5.6", "portable-net45+win8"]
    }
  },
  "buildOptions": {
    "emitEntryPoint": true,
    "preserveCompilationContext": true
  },
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true
    }
  },
  "publishOptions": {
    "include": ["wwwroot","web.config"]
  },
  "scripts": {
    "postpublish": [ "dotnet publish-iis --publish-folder
%publish:OutputPath% --framework %publish:FullTargetFramework%" ]
  }
}
```

Пакеты, добавленные в раздел `dependencies` файла `project.json`, предоставляют самую базовую функциональность, требующуюся для начала разработки приложения MVC. По мере разработки приложения SportsStore будут добавляться и другие пакеты, но пакеты в разделе `dependencies` являются хорошей отправной точкой (табл. 8.1).

Кроме пакетов в разделе dependencies в листинге 8.1 показано добавление в раздел tools файла project.json, которое конфигурирует пакет Microsoft.AspNetCore.Razor.Tools для применения в Visual Studio. Он включает средство IntelliSense для встроенных дескрипторных вспомогательных классов, используемых для создания HTML-содержимого, которое приспособлено под конфигурацию приложения MVC.

Таблица 8.1. Дополнительные пакеты NuGet в файле project.json

Имя	Описание
Microsoft.AspNetCore.Mvc	Этот пакет содержит инфраструктуру ASP.NET Core MVC и предоставляет доступ к основным средствам, таким как контроллеры и представления Razor
Microsoft.AspNetCore.StaticFiles	Этот пакет обеспечивает поддержку для обслуживания статических файлов, таких как файлы изображений, JavaScript и CSS, из папки wwwroot
Microsoft.AspNetCore.Razor.Tools	Этот пакет предлагает инструментальную поддержку для представлений Razor, в том числе средство IntelliSense для встроенных дескрипторных вспомогательных классов, которые применяются в представлениях внутри приложения SportsStore

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. 8.2, щелкните правой кнопкой мыши на элементе проекта SportsStore в окне Solution Explorer (элемент внутри папки src), выберите в контекстном меню пункт Add⇨New Folder (Добавить⇨Новая папка) и введите имя папки. Позже потребуются дополнительные папки, но создаваемые сейчас папки отражают главные части приложения MVC и для начала их вполне достаточно.

Таблица 8.2. Папки, требующиеся для проекта SportsStore

Имя	Описание
Models	Эта папка будет содержать классы моделей
Controllers	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

Приложение ASP.NET Core MVC полагается на несколько конфигурационных файлов. Имея установленные пакеты NuGet, понадобится отредактировать класс Startup, чтобы сообщить ASP.NET о том, что они должны использоваться (листинг 8.2).

Листинг 8.2. Включение средств в файле `Startup.cs`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace SportsStore {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app,
          IHostingEnvironment env, ILoggerFactory loggerFactory) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

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 метода `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. 8.3).

Таблица 8.3. Начальные методы для настройки средств, вызываемые в классе `Startup`

Метод	Описание
`UseDeveloperExceptionPage()`	Этот расширяющий метод отображает детали исключения, которое произошло в приложении, что полезно во время процесса разработки. Он не должен быть включен в развернутых приложениях; в главе 12 будет показано, как отключить данное средство
`UseStatusCodePages()`	Этот расширяющий метод добавляет простое сообщение в HTTP-ответы, которые иначе бы не имели тела, такие как ответы `404 - Not Found` (404 — не найдено)
`UseStaticFiles()`	Этот расширяющий метод включает поддержку для обслуживания статического содержимого из папки `wwwroot`
`UseMvcWithDefaultRoute()`	Этот расширяющий метод включает инфраструктуру ASP.NET Core MVC со стандартной конфигурацией (которая позже в процессе разработки будет изменена)

На заметку! Класс `Startup` — важное средство ASP.NET Core. Он будет подробно описан в главе 14.

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

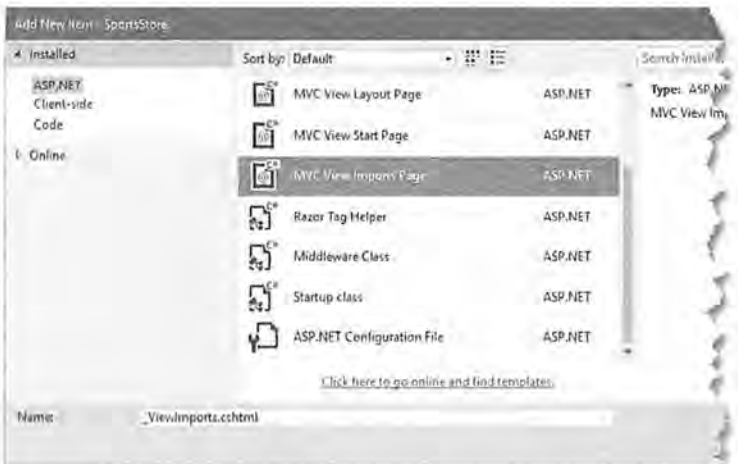

Рис. 8.3. Создание файла импортирования представлений

Щелкните на кнопке Add (Добавить), чтобы создать файл `_ViewImports.cshtml` и приведите его содержимое в соответствие с листингом 8.3.

Листинг 8.3. 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 дескрипторные вспомогательные классы, которые будут использоваться позже для создания элементов HTML, отражающих конфигурацию приложения SportsStore.

Создание проекта модульного тестирования

Создание проекта модульного тестирования требует выполнения процесса, который был описан в главе 7. С помощью проводника файлов создайте папку по имени `test` на том же уровне, что и существующая папка `src`, внутри папки решения `SportsStore`.

Возвратитесь в Visual Studio, щелкните правой кнопкой мыши на элементе решения `SportsStore` (элемент верхнего уровня в окне Solution Explorer), выберите в контекстном меню пункт Add⇨New Solution Folder (Добавить⇨Новая папка решения) и установите имя новой папки в `test`.

Щелкните правой кнопкой мыши на папке `test` в окне Solution Explorer и выберите в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект). Выберите шаблон Class Library (.NET Core) (Библиотека классов (.NET Core)) из категории Installed⇨Visual C#⇨.NET Core (Установленные⇨Visual C#⇨.NET Core), как показано на рис. 8.4, и установите имя проекта в `SportsStore.Tests`.

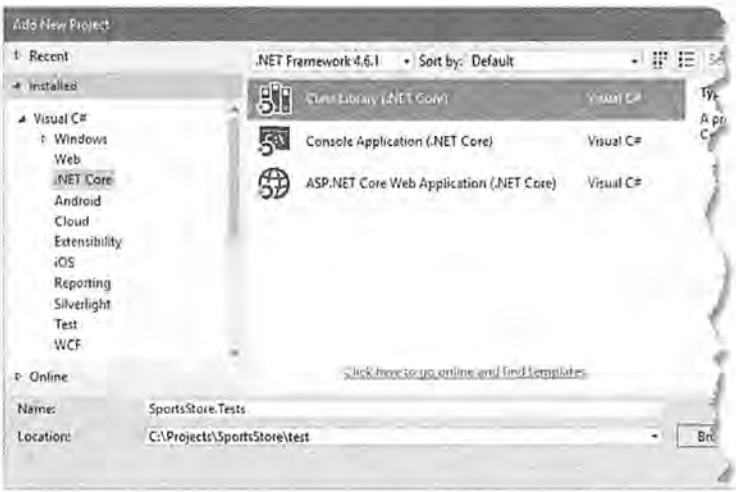

Рис. 8.4. Создание проекта модульного тестирования

Щелкните на кнопке Browse (Обзор) и перейдите в папку `test`. Щелкните на кнопке OK, чтобы выбрать эту папку, и затем щелкните на OK, чтобы создать проект модульного тестирования.

После создания проекта модульного тестирования отредактируйте содержащийся в нем файл `project.json` согласно листингу 8.4, и добавьте пакеты, необходимые для тестирования и создания имитированных объектов.

На заметку! Пакет `moq.netcore`, который применяется в листинге 8.4, требует изменения конфигурации Visual Studio, как было описано в разделе "Добавление инфраструктуры имитации" главы 7. Если вы не прорабатывали примеры из главы 7, то должны внести это изменение в конфигурацию прямо сейчас.

Листинг 8.4. Содержимое файла `project.json` из проекта модульного тестирования

```
{
  "version": "1.0.0-*",
  "testRunner": "xunit",
  "dependencies": {
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.0"
    },
    "xunit": "2.1.0",
    "dotnet-test-xunit": "2.2.0-preview2-build1029",
    "moq.netcore": "4.4.0-beta8",
```

```
    "System.Diagnostics.TraceSource": "4.0.0",
    "SportsStore": "1.0.0"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": ["dotnet5.6", "portable-net45+win8"]
    }
  }
}
```

Проверка и запуск приложения

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

Выбрав в меню Debug (Отладка) пункт Start Debugging (Запустить отладку) или Start Without Debugging (Запустить без отладки), если вы предпочитаете итеративный стиль разработки, описанный в главе 6, вы увидите страницу ошибки (рис. 8.6). Сообщение об ошибке отображается из-за того, что в настоящий момент в приложении нет контроллеров для обработки запросов; эту проблему мы вскоре решим.

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

Начало работы с моделью предметной области

Все проекты начинаются с модели предметной области, которая является центральной частью приложения MVC. Поскольку мы строим приложение электронной коммерции, то наиболее очевидная модель, которая необходима, касается товара. Добавьте в папку `Models` файл класса по имени `Product.cs` с определением, приведенным в листинге 8.5.

Листинг 8.5. 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; }
  }
}
```

Рис. 8.6. Выполнение приложения SportsStore

Создание хранилища

Нам нужен какой-то способ получения объектов `Product` из базы данных. Как объяснялось в главе 3, 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. Добавьте в папку `Models` новый файл интерфейса C# по имени `IProductRepository.cs` и поместите в него определение, показанное в листинге 8.6.

Листинг 8.6. Содержимое файла `IProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public interface IProductRepository {
    IEnumerable<Product> Products { get; }
  }
}
```

Этот интерфейс использует `IEnumerable<T>`, чтобы позволить вызывающему коду получать последовательность объектов `Product`, ничего не сообщая о том, как или где хранятся либо извлекаются данные. Класс, зависящий от интерфейса `IProductRepository`, может получать объекты `Product`, ничего не зная о том, откуда они поступают или каким образом класс реализации будет их доставлять. В процессе разработки мы еще будем возвращаться к интерфейсу `IProductRepository`, чтобы добавлять в него нужные средства.

Создание фиктивного хранилища

Теперь, когда определен интерфейс, можно было бы реализовать механизм постоянства 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, добавьте в папку `Models` файл класса по имени `FakeProductRepository.cs` и поместите в него определение, представленное в листинге 8.7.

Листинг 8.7. Содержимое файла `FakeProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public class FakeProductRepository : IProductRepository {
```

```
    public IEnumerable<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 }
    };
  }
}
```

Класс `FakeProductRepository` реализует интерфейс `IProductRepository`, возвращая фиксированную коллекцию объектов `Product` в качестве значения свойства `Products`.

Регистрация службы хранилища

В MVC делается особый акцент на применении *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*, которые предоставляют функциональные средства, используемые другими частями приложения. Класс, предоставляющий службу, впоследствии может быть модифицирован или заменен, не требуя внесения изменений в классы, которые его задействуют. Более подробно это объясняется в главе 18, а в случае приложения SportsStore необходимо создать службу хранилища, которая позволит контроллерам получать реализующие интерфейс `IProductRepository` объекты, не зная, какой класс применяется. В итоге появится возможность начать разработку приложения с использованием простого класса `FakeProductRepository`, созданного в предыдущем разделе, и позже заменить его реальным хранилищем, не внося изменения во все классы, которым нужен доступ в хранилище. Службы регистрируются в методе `ConfigureServices()` класса `Startup`; в листинге 8.8 определена новая службы для хранилища.

Листинг 8.8. Создание службы хранилища в файле `Startup.cs`

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

namespace SportsStore {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IProductRepository, FakeProductRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env, ILoggerFactory loggerFactory) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Добавленный в метод `ConfigureServices()` оператор сообщает инфраструктуре ASP.NET о том, что когда компоненту наподобие 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

Оставшуюся часть 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 применению MVC, вернувшись к добавлению средств модели и хранилища, когда они понадобятся.

В этом 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 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. Выбор таких пунктов меню способствует выбору сценария для создаваемого компонента, такого как контроллер с действиями, допускающими чтение/запись, или представление, которое содержит форму, применяемую для создания специфического объекта модели.

В настоящей книге формирование шаблонов не используется. Код и разметка, генерируемые средством формирования шаблонов, являются настолько общими, что едва ли не бесполезны, 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 средство формирования шаблонов.

Тем не менее, это еще одна ситуация, когда ваш стиль разработки может отличаться от моего, и вы вполне можете предпочесть работу со средством формирования шаблонов. В таком случае можете включить его, сделав ряд добавлений в файле `project.json`. Во-первых, в разделе `dependencies` потребуется указать два новых пакета:

```
...
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.0",
    "type": "platform"
  },
```

```
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",
    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.0",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    },
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0",
    "Microsoft.VisualStudio.Web.CodeGeneration.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    },
    "Microsoft.VisualStudio.Web.CodeGenerators.Mvc": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    }
  },
  ...
```

Во-вторых, эти пакеты должны быть зарегистрированы в разделе `tools`:

```
  ...
  "tools": {
    "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
    "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final",
    "Microsoft.VisualStudio.Web.CodeGeneration.Tools": {
      "version": "1.0.0-preview2-final",
      "imports": [
        "portable-net45+win8+dnxcore50",
        "portable-net45+win8"
      ]
    }
  },
  ...
```

После сохранения изменений и установки пакетов средой Visual Studio вы увидите новые пункты в контекстных меню, открываемых по щелчку правой кнопкой мыши на папках в окне Solution Explorer. Выбор этих пунктов меню будет приводить к появлению диалоговых окон, позволяющих выбирать сценарии, которые должны применяться для создания контроллера или представления.

Добавление контроллера

Чтобы создать первый контроллер в приложении, добавьте в папку `Controllers` файл класса по имени `ProductController.cs` с определением, показанным в листинге 8.9.

Листинг 8.9. 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, она 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 необходимо применять класс `FakeRepository`, а также о том, что каждый раз должен создаваться 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, то рекомендую вам дождаться главы 18, прежде чем принимать решение.

Далее добавьте метод действия по имени `List()`, который будет визуализировать представление, отображающее полный список товаров из хранилища (листинг 8.10).

Листинг 8.10. Добавление метода действия в файле `ProductController.cs`

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

namespace SportsStore.Controllers {
  public class ProductController : Controller {
    private IProductRepository repository;
    public ProductController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult List() => View(repository.Products);
  }
}
```

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

Добавление и конфигурирование представления

Нам нужно создать представление для отображения содержимого пользователю, но потребуется проделать ряд подготовительных шагов, чтобы упростить написание представления. Сначала необходимо 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.

Создайте папку `Views/Shared` и добавьте в нее новую страницу компоновки представлений MVC по имени `_Layout.cshtml`, которое является стандартным именем, назначаемым средой Visual Studio элементу такого типа. В листинге 8.11 приведено содержимое файла `_Layout.cshtml`. В стандартное содержимое внесено одно изменение, связанное с установкой внутренностей элемента `title` в `SportsStore`.

Листинг 8.11. 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>
```

Далее понадобится сконфигурировать приложение, чтобы файл `_Layout.cshtml` применялся по умолчанию. Это делается добавлением в папку `Views` файла с шаблоном MVC View Start Page (Файл запуска представления MVC) и именем `_ViewStart.cshtml`.

Стандартное содержимое, добавляемое Visual Studio (листинг 8.12), выбирает компоновку по имени `_Layout.cshtml`, которая соответствует файлу из листинга 8.11.

Листинг 8.12. Содержимое файла `_ViewStart.cshtml` из папки `Views`

```
@{
  Layout = "_Layout";
}
```

Теперь необходимо добавить представление, которое будет отображаться, когда для обработки запроса используется метод действия `List()`. Создайте папку `Views/Product` и добавьте в нее файл представления Razor по имени `List.cshtml`. Поместите в него разметку, показанную в листинге 8.13.

Листинг 8.13. 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 и генерация простого набора HTML-элементов для каждого полученного объекта `Product`.

Представлению не известно, откуда поступили объекты `Product`, как они были получены или охватывают ли они все товары, известные приложению. Взамен представление имеет дело только с тем, как отображать детали 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`, тогда вызов `(1002.3).ToString("c")` возвратит значение `$1,002.30`, но если для сервера установлена культура `en-GB`, то этот же вызов возвратит значение `£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`. Это делается путем редактирования оператора в классе `Startup`, который настраивает классы MVC, обрабатывающие HTTP-запросы (листинг 8.14).

Листинг 8.14. Изменение стандартного маршрута в файле `Startup.cs`

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;
namespace SportsStore {
  public class Startup {
    public void ConfigureServices(IServiceCollection services) {
      services.AddTransient<IProductRepository,
      FakeProductRepository>();
      services.AddMvc();
    }
```

```
    public void Configure(IApplicationBuilder app,
            IHostingEnvironment env, ILoggerFactory loggerFactory) {
      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 8.14, указывают инфраструктуре MVC на необходимость отправки запросов методу действия `List()` контроллера `Product`, если только в URL запроса не указано иное.

Совет. Обратите внимание, что в листинге 8.14 имя контроллера указано как `Product`, а не `ProductController`, являющееся именем класса. Это часть соглашения об именовании MVC, в рамках которого имена классов 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 товара. При генерации HTML-ответа инфраструктура MVC объединит данные из представления, выбранные методом действия, с содержимым разделяемой компоновки, порождая завершенный HTML-документ, который браузер в состоянии разобрать и отобразить. Запустив приложение, можно увидеть результат, показанный на рис. 8.7.

Это типовой шаблон разработки для инфраструктуры ASP.NET Core MVC. Начальные затраты времени на необходимую настройку являются обязательными, но затем базовые 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 подойдет для ваших проектов.

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

Продукты Visual Studio и SQL Server располагают удобным средством *LocalDB*, которое представляет собой не требующую администрирования реализацию основной функциональности SQL Server, специально спроектированную для разработчиков. Благодаря этому средству 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, а разработчики приложений могут заниматься написанием кода.

Совет. Если при установке Visual Studio вы не выбрали LocalDB, то придется сделать это сейчас. Средство представляет собой часть инструментов для работы с данными или же его можно установить как часть SQL Server.

Установка Entity Framework Core

Инфраструктура Entity Framework Core устанавливается с применением NuGet, а в листинге 8.15 показаны требуемые добавления в раздел `dependencies` файла `project.json` в проекте `SportsStore`.

Листинг 8.15. Добавление Entity Framework Core в файле `project.json` внутри проекта `SportsStore`

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

Базы данных управляются с использованием инструментов командной строки, которые настраиваются в разделе `tools` файла `project.json` (листинг 8.16).

Листинг 8.16. Регистрация инструментов EF Core в файле `project.json` внутри проекта `SportsStore`

```
...
"tools": {
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final",
  "Microsoft.EntityFrameworkCore.Tools": {
    "version": "1.0.0-preview2-final",
    "imports": [ "portable-net45+win8+dnxcore50", "portable-net45+win8" ]
  }
},
...
```

После сохранения файла `project.json` среда Visual Studio загрузит и установит инфраструктуру EF Core, а также добавит ее в проект.

Создание классов базы данных

Класс контекста базы данных является шлюзом между приложением и EF Core, обеспечивая доступ к данным приложения с применением объектов моделей. Чтобы создать класс контекста базы данных для приложения SportsStore, добавьте в папку `Models` файл класса по имени `ApplicationDbContext.cs` с определением, приведенным в листинге 8.17.

Листинг 8.17. Содержимое файла `ApplicationDbContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

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. Для наполнения базы данных добавьте в папку `Models` файл класса по имени `SeedData.cs` и поместите в него определение, показанное в листинге 8.18.

Листинг 8.18. Содержимое файла `SeedData.cs` из папки `Models`

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

namespace SportsStore.Models {

  public static class SeedData {

    public static void EnsurePopulated(IApplicationBuilder app) {
      ApplicationDbContext context = app.ApplicationServices
        .GetRequiredService<ApplicationDbContext>();
      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`, который является классом, используемым в методе `Configure()` класса `Startup` при регистрации классов промежуточного программного обеспечения для обработки HTTP-запросов; именно здесь будет обеспечиваться наличие содержимого в базе данных.

Метод The `EnsurePopulated()` получает объект `ApplicationDbContext` посредством интерфейса `IApplicationBuilder` и применяет его для проверки, присутствуют ли в базе данных какие-нибудь объекты `Product`. Если объектов нет, то база данных наполняется с использованием коллекции объектов `Product` и метода `AddRange()`, после чего сохраняется с помощью метода `SaveChanges()`.

Создание класса хранилища

Хотя в настоящий момент 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 применением инфраструктуры Entity Framework Core. Добавьте в папку `Models` файл класса по имени `EFProductRepository.cs` с определением класса хранилища, представленным в листинге 8.19.

Листинг 8.19. Содержимое файла EFProductRepository.cs из папки Models

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
      context = ctx;
    }
    public IEnumerable<Product> Products => context.Products;
  }
}
```

По мере добавления средств к приложению класс будет дополняться новой функциональностью, но пока что реализация хранилища просто отображает свойство `Products`, определенное в интерфейсе `IProductRepository`, на свойство `Products`, которое определено в классе `ApplicationDbContext`.

Определение строки подключения

Строка подключения указывает местоположение и имя базы данных, а также предоставляет конфигурационные настройки, с которыми приложение должно подключаться 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 ASP.NET диалогового окна Add New Item.

При создании файла `appsettings.json` среда Visual Studio добавляет в него заполнитель для строки подключения, который необходимо привести в соответствие с листингом 8.20.

Листинг 8.20. Редактирование строки подключения в файле appsettings.json

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

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

Совет. Строка подключения должна быть выражена в виде единственной неразрывной строки кода, что нормально для редактора Visual Studio, но не умещается на печатной странице и приводит к неуклюжему форматированию в листинге 8.20. При определении строки подключения в собственном проекте удостоверьтесь, что значение элемента `ConnectionString` находится в единственной строке кода.

Конфигурирование приложения

Далее понадобится прочитать строку подключения и сконфигурировать приложение для ее использования при подключении к базе данных. Для чтения строки подключения из файла `appsettings.json` требуется еще один пакет NuGet. В листинге 8.21 показано изменение раздела `dependencies` внутри файле `project.json`.

Листинг 8.21. Добавление пакета в файле `project.json` проекта `SportsStore`

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

Этот пакет делает возможным чтение конфигурационных данных из файлов JSON, таких как `appsettings.json`. Чтобы применить функциональность, предлагаемую новым пакетом, для чтения строки подключения из конфигурационного файла и чтобы настроить EF Core, в класс `Startup` необходимо внести соответствующее изменение (листинг 8.22).

Листинг 8.22. Конфигурирование приложения в файле `Startup.cs`

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

namespace SportsStore {

  public class Startup {
    IConfigurationRoot Configuration;

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

```
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json").Build();
    }
    public void ConfigureServices(IServiceCollection services) {
      services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
          Configuration["Data:SportStoreProducts:ConnectionString"]));
      services.AddTransient<IProductRepository, EFProductRepository>();
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app,
          IHostingEnvironment env, ILoggerFactory loggerFactory) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvc(routes => {
        routes.MapRoute(
          name: "default",
          template: "{controller=Product}/{action=List}/{id?}");
      });
      SeedData.EnsurePopulated(app);
    }
  }
}
```

Добавленный в класс `Startup` конструктор загружает конфигурационные настройки из файла `appsettings.json` и делает их доступными через свойство по имени `Configuration`. Особенности чтения и доступа к конфигурационным данным рассматриваются в главе 14.

В метод `ConfigureServices()` добавлена последовательность обращений к методам, которая настраивает инфраструктуру Entity Framework Core.

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

Расширяющий метод `AddDbContext()` настраивает службы, предоставляемые инфраструктурой Entity Framework Core для класса контекста базы данных, который был создан в листинге 8.17. Как объяснялось в главе 14, многие методы, используемые в классе `Startup`, позволяют конфигурировать службы и средства промежуточного программного обеспечения с применением аргументов с параметрами. 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 одно изменение, внесенное в класс `Startup`, было связано с заменой фиктивного хранилища реальным:

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

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

Финальное изменение в классе `Startup` касается вызова метода `SeedData.EnsurePopulated()`, который гарантирует наличие в базе данных определенной тестовой информации и вызывается внутри метода `Configure()` класса `Startup`. При запуске приложения метод `Startup.ConfigureServices()` вызывается перед методом `Startup.Configure()`, т.е. ко времени вызова метода `SeedData.EnsurePopulated()` можно иметь уверенность в том, что службы Entity Framework Core уже установлены и сконфигурированы.

Создание и применение миграции базы данных

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

Команды EF Core выполняются с применением консоли диспетчера пакетов (Package Manager Console), которая открывается через пункт меню Tools⇨NuGet Package Manager (Сервис⇨Диспетчер пакетов NuGet) в Visual Studio.

Запустите в консоли диспетчера пакетов следующие команды, чтобы создать класс миграции, который подготовит базу данных к первому использованию:

```
Add-Migration Initial
```

Когда команда завершит свое выполнение, вы увидите в окне Solution Explorer среды Visual Studio папку `Migrations`. Именно в ней инфраструктура Entity Framework Core хранит классы миграции. Одно из имен файлов будет выглядеть как длинное число с `_Initial.cs` после него, и это класс, который будет применяться для создания начальной схемы базы данных. Просмотрев содержимое такого файла, вы увидите, как класс модели `Product` использовался для создания схемы.

Запустите приведенную ниже команду, чтобы создать базу данных и выполнить команды миграции:

```
Update-Database
```

Создание базы данных займет какое-то время, но после завершения работы команды вы сможете увидеть результат, запустив приложение. 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 EF Core, которая загружает реляционные 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).

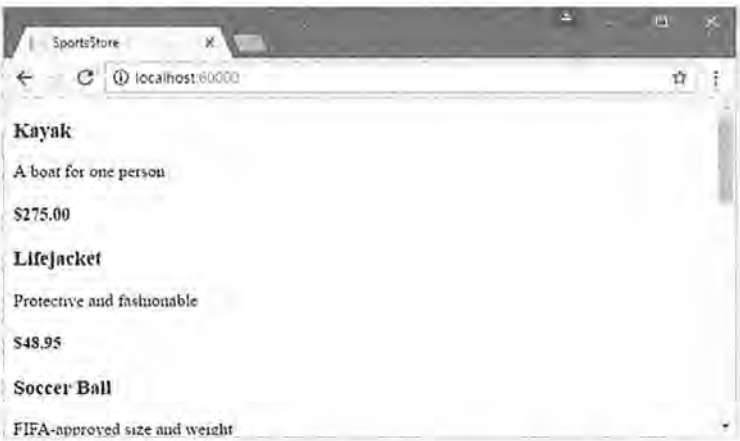

Рис. 8.8. Использование хранилища в виде базы данных

Такой подход с применением Entity Framework Core для представления базы данных SQL Server в виде последовательности объектов моделей отличается простотой и легкостью, позволяя сосредоточить все внимание на инфраструктуре ASP.NET Core MVC. Я опустил множество деталей, связанных с функционированием 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, находящийся по адресу http://ef.readthedocs.io.

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 каталог. В метод действия List() контроллера Product необходимо добавить параметр, как показано в листинге 8.23.

Листинг 8.23. Добавление поддержки разбиения на страницы в метод действия List() в файле ProductController.cs

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

Поле `PageSize` указывает, что на одной странице должны отображаться сведения о четырех товарах. Позже мы заменим его более совершенным механизмом. В метод `List()` добавлен необязательный параметр. Это означает, что в случае вызова метода без параметра (`List()`) вызов обрабатывается 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 запрашивать конкретную страницу. Затем полученные объекты `Product` можно сравнить с теми, которые ожидались 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"}
      });
      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, в конец URL можно добавить параметр строки запроса, например:

```
http://localhost:60000/?page=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 этих 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. Для этого мы реализуем *дескрипторный вспомогательный класс*, который будет генерировать HTML-разметку для требуемых ссылок.

Добавление модели представления

Чтобы обеспечить поддержку дескрипторного вспомогательного класса, 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. Создайте в проекте `SportsStore` папку `Models/ViewModels` и добавьте в нее файл класса с содержимым, приведенным в листинге 8.24.

Листинг 8.24. 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 дескрипторного вспомогательного класса

Теперь, имея модель представления, можно создать дескрипторный вспомогательный класс. Создайте в проекте `SportsStore` папку `Infrastructure` и добавьте в нее файл класса по имени `PageLinkTagHelper.cs` с определением, показанным в листинге 8.25. Дескрипторные вспомогательные классы 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 8.25. 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 { page = i });
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
      }
      output.Content.AppendHtml(result.InnerHtml);
    }
  }
}
```

Этот дескрипторный вспомогательный класс заполняет элемент `div` элементами `a`, которые соответствуют страницам товаров. Сейчас мы не будем вдаваться в какие-либо детали относительно дескрипторных вспомогательных классов; достаточно знать, что они предлагают один из наиболее удобных способов помещения логики C# в представления. Код для дескрипторного вспомогательного класса может выглядеть запутанным, потому что смешивать C# и HTML непросто. Но использование дескрипторных вспомогательных классов является предпочтительным способом включения блоков кода C# в представление, поскольку дескрипторный вспомогательный класс можно легко подвергать модульному тестированию.

Большинство компонентов MVC, таких как контроллеры и представления, обнаруживаются автоматически, но дескрипторные вспомогательные классы должны быть зарегистрированы. В листинге 8.26 приведено содержимое файла `_ViewImports.cshtml` из папки `Views` с добавленным оператором, который сообщает MVC о том, что дескрипторные вспомогательные классы 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 8.26. Регистрация дескрипторного вспомогательного класса в файле `_ViewImports.cshtml`

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

Модульное тестирование: создание ссылок на страницы

Чтобы протестировать дескрипторный вспомогательный класс `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 этого теста связана с созданием объектов, которые требуются для создания и применения дескрипторного вспомогательного класса. Дескрипторные вспомогательные классы используют объекты `IUrlHelperFactory` для генерации URL, которые указывают на разные части приложения, и для создания реализаций этого интерфейса и связанного с ним интерфейса `IUrlHelper`, предоставляющего тестовые данные, используется инфраструктура Moq.

Основная часть теста проверяет вывод дескрипторного вспомогательного класса 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 дескрипторный вспомогательный класс, 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. Добавьте в папку `Models/ViewModels` проекта `SportsStore` файл класса по имени `ProductsListViewModel.cs` с содержимым, приведенным в листинге 8.27.

Листинг 8.27. 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; }
  }
}
```

Теперь можно обновить метод действия `List()` класса `ProductController` так, чтобы он использовал класс `ProductsListViewModel` для снабжения представления сведениями о товарах, отображаемых на страницах, и информацией о разбиении на страницы (листинг 8.28).

Листинг 8.28. Обновление метода действия `List()` в файле `ProductController.cs`

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;
namespace SportsStore.Controllers {
  public class ProductController : Controller {
    private IProductRepository repository;
    public int PageSize = 4;
    public ProductController(IProductRepository repo) {
      repository = repo;
    }
    public ViewResult List(int page = 1)
      => View(new ProductsListViewModel {
        Products = repository.Products
          .OrderBy(p => p.ProductID)
          .Skip((page - 1) * PageSize)
          .Take(PageSize),
        PagingInfo = new PagingInfo {
          CurrentPage = page,
          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"}
  });

  // 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"}
  });
  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 этой, 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 8.29, чтобы иметь дело с новым типом модели представления.

Листинг 8.29. Обновление файла `List.cshtml`

```
@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, 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 только добавить HTML-элемент, который дескрипторный вспомогательный класс будет обрабатывать для создания ссылок на страницы (листинг 8.30).

Листинг 8.30. Добавление ссылок на страницы в файле `List.cshtml`

```
@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.

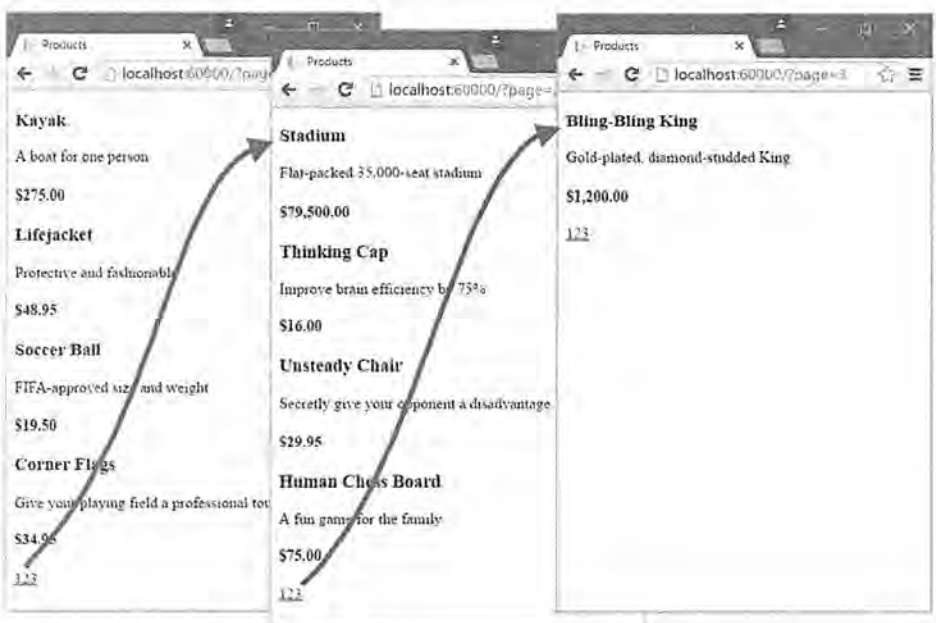

Рис. 8.9. Отображение ссылок для навигации по страницам

На заметку! Если вы запустите приложение с применением пункта меню Start Debugging (Запустить отладку), то можете столкнуться с сообщением об ошибке, предупреждающим о том, что коллекция была модифицирована. Это дефект EF Core, который должен быть исправлен к моменту выхода настоящей книги, но если он все же останется, тогда простая перезагрузка окна браузера решит проблему и приведет к отображению содержимого, представленного на рис. 8.9.

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 задействует подходящее 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, по ходу дела мы создаем модульные тесты, что позволяет проверять поведение приложения естественным образом, который практически невозможен в случае применения сложного элемента управления Web Forms. Наконец, 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, подобную следующей:

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

Чтобы получить более привлекательные URL, необходимо создать схему, которая следует шаблону *компонуемых URL*. Компонуемый URL — это 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 8.31).

Листинг 8.31. Добавление нового маршрута в файле `Startup.cs`

```
...
public void Configure(IApplicationBuilder app,
      IHostingEnvironment env, ILoggerFactory loggerFactory) {
  app.UseDeveloperExceptionPage();
  app.UseStatusCodePages();
```

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

Важно поместить этот маршрут 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.

Это единственное изменение, которое потребуется внести, чтобы изменить схему URL для разбиения на страницы списка товаров. Инфраструктура MVC тесно интегрирована с функцией маршрутизации, поэтому приложение автоматически отражает изменение подобного рода в URL, используемых приложением, что касается и URL, которые генерируются дескрипторными вспомогательными классами 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, вы увидите 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 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 это умаляет его технические достоинства. В настоящем разделе мы постараемся исправить ситуацию. Мы собираемся реализовать классическую компоновку, содержащую два столбца и заголовок, как показано на рис. 8.11.

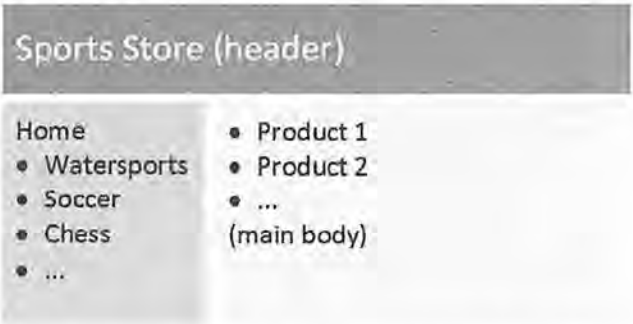

Рис. 8.11. Целевой дизайн для приложения SportsStore

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. Выберите шаблон элемента Bower Configuration File (Файл конфигурации Bower) из категории Client-side (Клиентская сторона) в диалоговом окне Add New Item, чтобы создать в проекте `SportsStore` файл по имени `bower.json`, как демонстрировалось в главе 6. Добавьте в раздел `dependencies` созданного файла пакет Bootstrap, как показано в листинге 8.32.

Листинг 8.32. Добавление пакета Bootstrap в файле `bower.json` внутри проекта `SportsStore`

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6"
  }
}
```

После сохранения внесенных в файл `bower.json` изменений среда Visual Studio применяет инструмент Bower для загрузки пакета 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 было показано, как работают представления Razor, как они используются и как встраивать в них компоновки. Файл запуска представления, добавленный в начале главы, указывает, что файл по имени `_Layout.cshtml` должен выступать в качестве стандартной компоновки, так что начальную стилизацию Bootstrap мы применим именно к нему (листинг 8.33).

Листинг 8.33. 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/css/*.min.css" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse" role="navigation">
    <a class="navbar-brand" href="#">SPORTS STORE</a>
  </div>
  <div class="row panel">
    <div id="categories" class="col-xs-3">
      Put something useful here later
    </div>
    <div class="col-xs-8">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Элемент `link` имеет атрибут `asp-href-include`, который представляет собой пример использования встроенного дескрипторного вспомогательного класса. В данном случае дескрипторный вспомогательный класс просматривает значение атрибута и генерирует элементы `link` для всех файлов по указанному пути, который может содержать групповые символы. Это удобное средство гарантирования того, что вы можете добавлять и удалять файлы из структуры папок `wwwroot`, не нарушая работу приложения, но, как объясняется в главе 25, указание групповых символов требует особого внимания, чтобы они соответствовали нужным файлам.

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 8.34 стилизация используется в файле `List.cshtml`.

Листинг 8.34. Стилизация содержимого в файле List.cshtml

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <div class="well">
    <h3>
      <strong>@p.Name</strong>
      <span class="pull-right label label-primary">
        @p.Price.ToString("c")
      </span>
    </h3>
    <span class="lead">@p.Description</span>
  </div>
}
```

```
<div page-model="@Model.PagingInfo" page-action="List"
     page-classes-enabled="true"
     page-class="btn" page-class-normal="btn-default"
     page-class-selected="btn-primary"class="btn-group pull-right">
</div>
```

Нам необходимо стилизовать кнопки, которые генерирует класс `PageLinkTagHelper`, но жестко встраивать классы Bootstrap в код нежелательно, т.к. это затруднит повторное использование дескрипторного вспомогательного класса где-то в другом месте приложения либо изменение внешнего вида кнопок. Взамен мы определяем специальные атрибуты в элементе `div`, которые указывают требуемые классы. Данные атрибуты соответствуют свойствам, добавленным в дескрипторный вспомогательный класс, которые затем применяются для стилизации создаваемых элементов `a` (листинг 8.35).

Листинг 8.35. Добавление классов к генерируемым элементам в файле PageLinkTagHelper.cs

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

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 { page = 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 в дескрипторном вспомогательном классе, учитывая отображение между форматом имени атрибута HTML (`page-class-normal`) и форматом имени свойства C# (`PageClassNormal`). Это позволяет дескрипторным вспомогательным классам поразному реагировать в зависимости от атрибутов HTML-элемента, обеспечивая более гибкий способ генерации содержимого в приложении MVC.

Запустив приложение, вы увидите, 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 данной 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, добавьте в папку `Views/Shared` файл представления Razor по имени `ProductSummary.cshtml` и поместите в него разметку, показанную в листинге 8.36.

Листинг 8.36. Содержимое файла `ProductSummary.cshtml` из папки `Views/Shared`

```
@model Product

<div class="well">
  <h3>
    <strong>@Model.Name</strong>
    <span class="pull-right label label-primary">
      @Model.Price.ToString("c")
    </span>
  </h3>
  <span class="lead">@Model.Description</span>
</div>
```

Теперь необходимо модифицировать файл `List.cshtml` из папки `Views/Products`, чтобы в нем применялось частичное представление (листинг 8.37).

Листинг 8.37. Использование частичного представления в файле `List.cshtml`

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  @Html.Partial("ProductSummary", p)
}

<div page-model="@Model.PagingInfo" page-action="List"
     page-classes-enabled="true"
     page-class="btn" page-class-normal="btn-default"
     page-class-selected="btn-primary"class="btn-group pull-right">
</div>
```

Мы взяли код разметки, который ранее размещался в цикле `foreach` в представлении `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.

Резюме

В настоящей главе была построена основная инфраструктура для приложения SportsStore. Пока что она не содержит достаточного набора функциональных средств, чтобы их прямо сейчас можно было продемонстрировать пользователю, 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 `ProductController`, который может создавать разбитые на страницы списки товаров, а также настроена ясная и дружественная к пользователям схема URL.

Рис. 8.13. Использование частичного представления

В этой 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.