Администрирование информационных систем

02.03.03 - Математическое обеспечение и администрирование информационных систем, направленность (профиль) - разработка и администрирование информационных систем

http//vikchas.ru

Лабораторные работы

Работа 8 Тема «Администрирование и защита ИС»

Часовских Виктор Петрович
доктор технических наук,
Профессор кафедры ШИиКМ
ФГБОУ ВО «Уральский государственный
экономический университет

Екатеринбург 2024

SportsStore: администрирование

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

Управление заказами

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

Расширение модели

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

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

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

namespace SportsStore.Models {

  public class Order {

    [BindNever]
    public int OrderID { get; set; }
    [BindNever]
    public ICollection<CartLine> Lines { get; set; }

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

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

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

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

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

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

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

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

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

```
dotnet ef migrations add ShippedOrders
```

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

Добавление действий и представления

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

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

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

namespace SportsStore.Controllers {

  public class OrderController : Controller {
    private IOrderRepository repository;
```

```
    private Cart cart;

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

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

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

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

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

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

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

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

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

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

```
@model IEnumerable<Order>

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

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

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

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

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

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

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

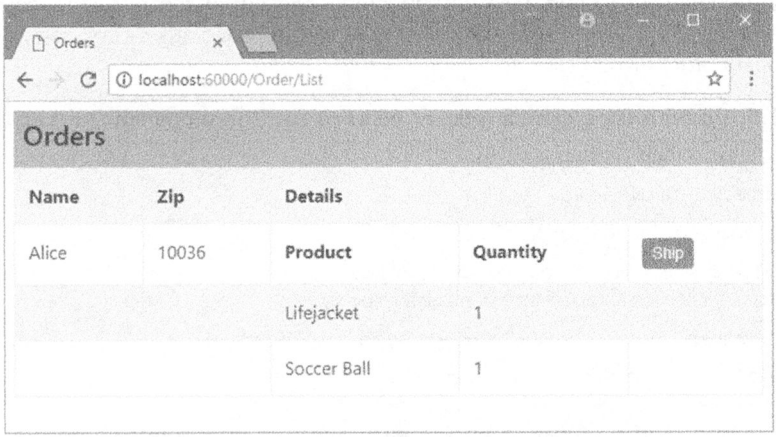

Рис. 11.1. Управление заказами

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

Добавление средств управления каталогом

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

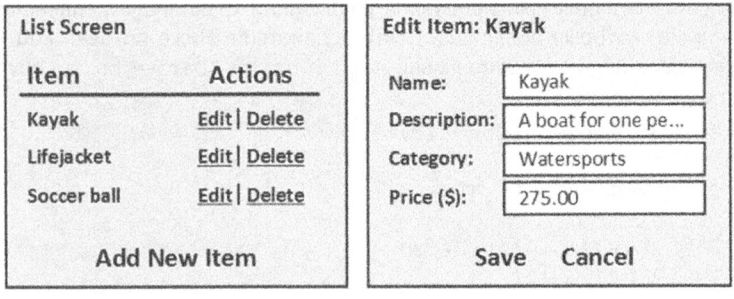

Рис. 11.2. Эскиз пользовательского интерфейса CRUD для каталога товаров

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

Создание контроллера CRUD

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

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

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

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

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

Модульное тестирование: метод действия `Index()`

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

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

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

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

Реализация представления списка

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

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

```
@model IEnumerable<Product>

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

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

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

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

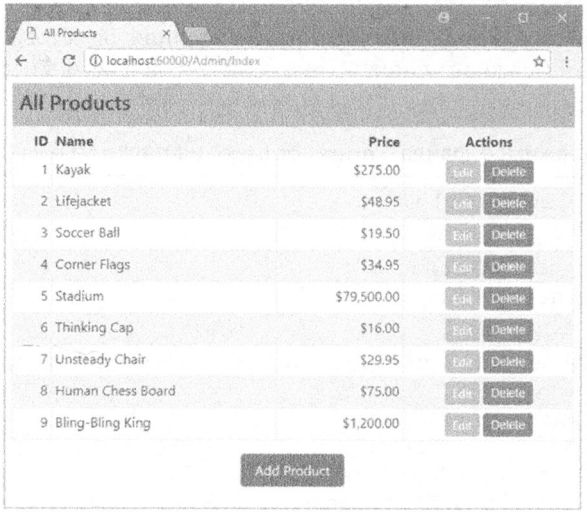

Рис. 11.3. Отображение списка товаров

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

Редактирование сведений о товарах

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

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

Создание метода действия `Edit()`

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

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

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

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

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

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

Модульное тестирование: метод действия Edit()

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

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

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

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

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

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

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

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

Создание представления редактирования

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

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

```
@model Product

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

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

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

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

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

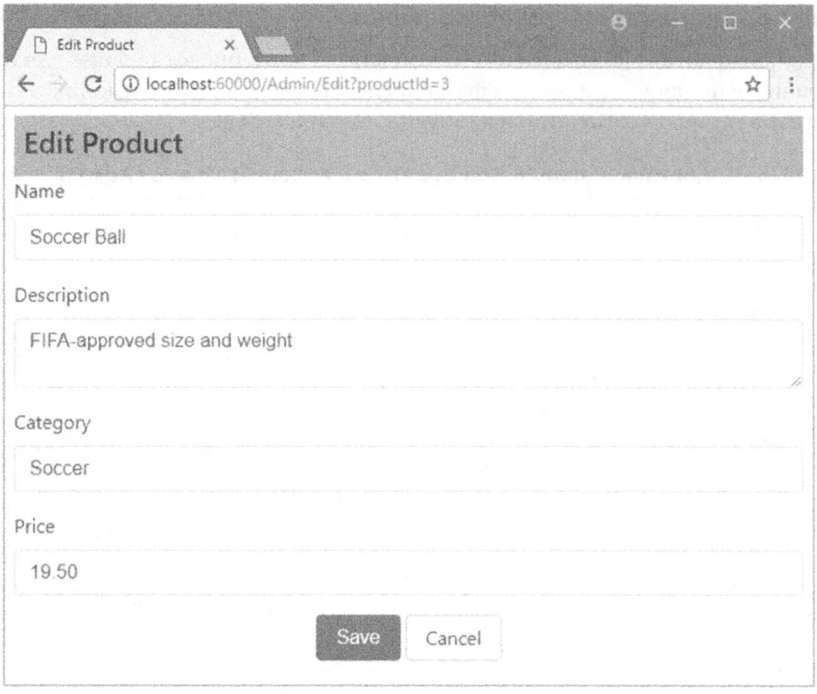

Рис. 11.4. Отображение сведений о товаре для редактирования

Обновление хранилища товаров

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

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

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

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

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

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

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

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

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

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

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

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

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

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

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

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

namespace SportsStore.Models {

  public class FakeProductRepository /* : IProductRepository */ {

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

Обработка запросов POST в методе действия Edit()

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

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

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

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

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

    [HttpPost]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Отображение подтверждающего сообщения

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

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

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

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

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

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

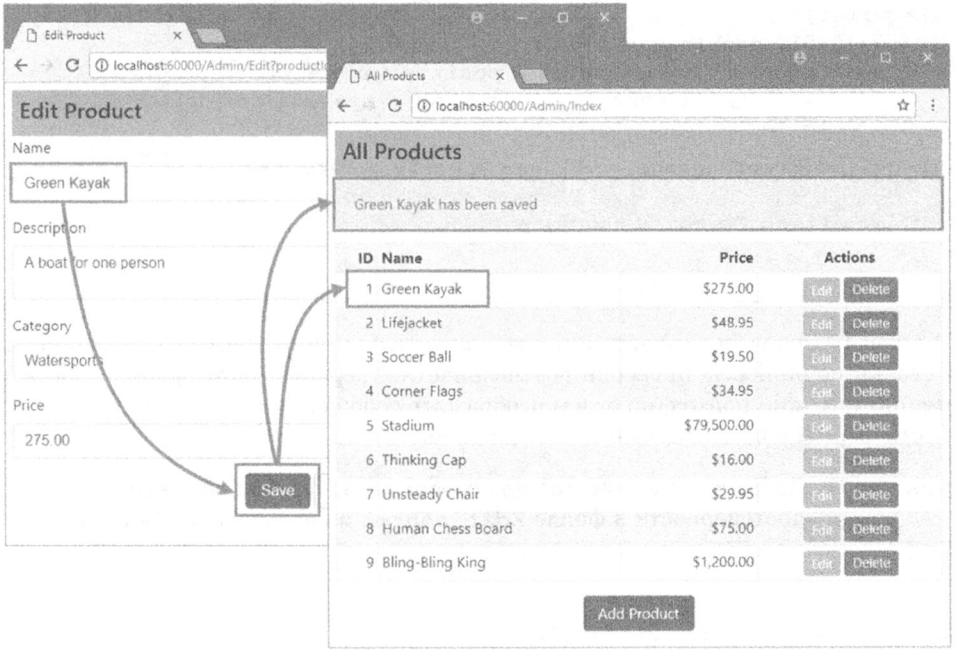

Рис. 11.5. Редактирование сведений о товаре и отображение сообщения из `TempData`

Добавление проверки достоверности модели

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

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

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

namespace SportsStore.Models {

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

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

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

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

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

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

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

```
@model Product

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

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

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

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

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

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

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

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

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

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

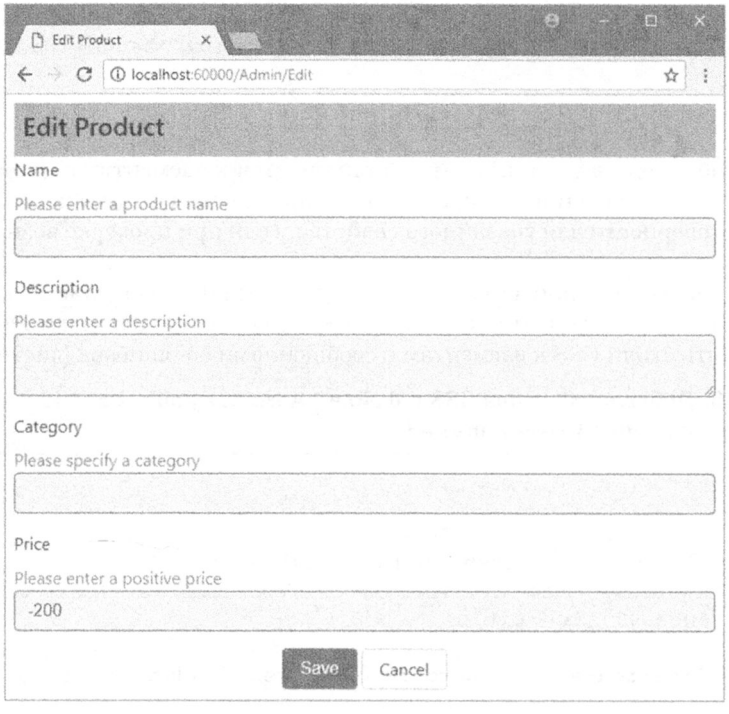

Рис. 11.6. Проверка достоверности данных при редактировании сведений о товаре

Включение проверки достоверности на стороне клиента

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

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

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

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

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

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

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

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

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

Создание новых товаров

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

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

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

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

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

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

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

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

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

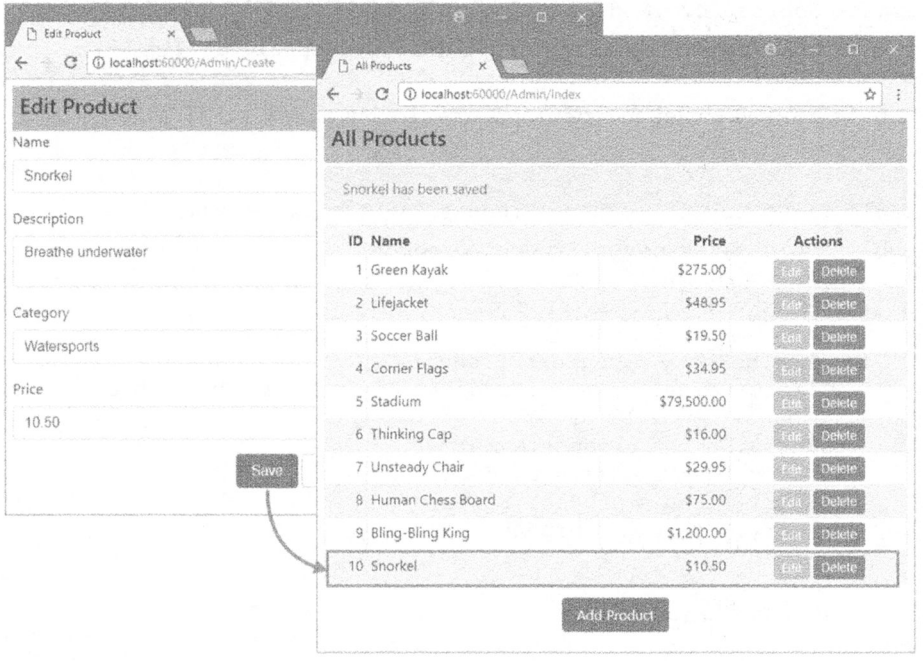

Рис. 11.7. Добавление нового товара в каталог

Удаление товаров

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

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

```
using System.Linq;

namespace SportsStore.Models {

  public interface IProductRepository {

    IQueryable<Product> Products { get; }

    void SaveProduct(Product product);

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

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

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

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

namespace SportsStore.Models {

  public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

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

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

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

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

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

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

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

namespace SportsStore.Controllers {

  public class AdminController : Controller {
    private IProductRepository repository;

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

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

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

    [HttpPost]
    public IActionResult Edit(Product product) {

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

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

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

Модульное тестирование: удаление товаров

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

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

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

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

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

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

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

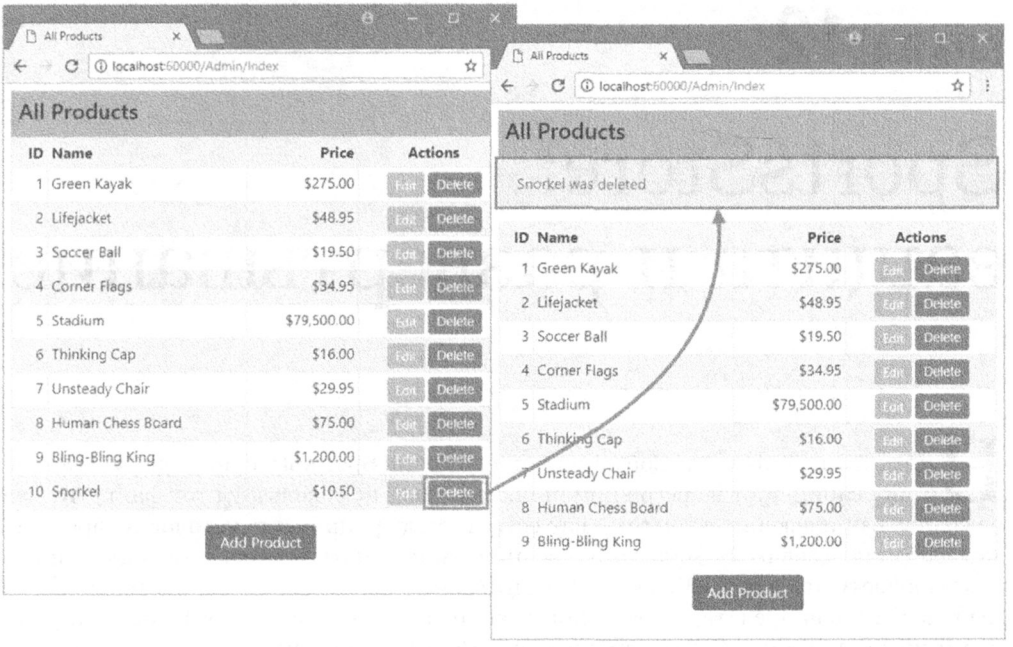

Рис. 11.8. Удаление товара из каталога

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

Резюме

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

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 метапакет, 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 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();
  }
 }
}
```

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.

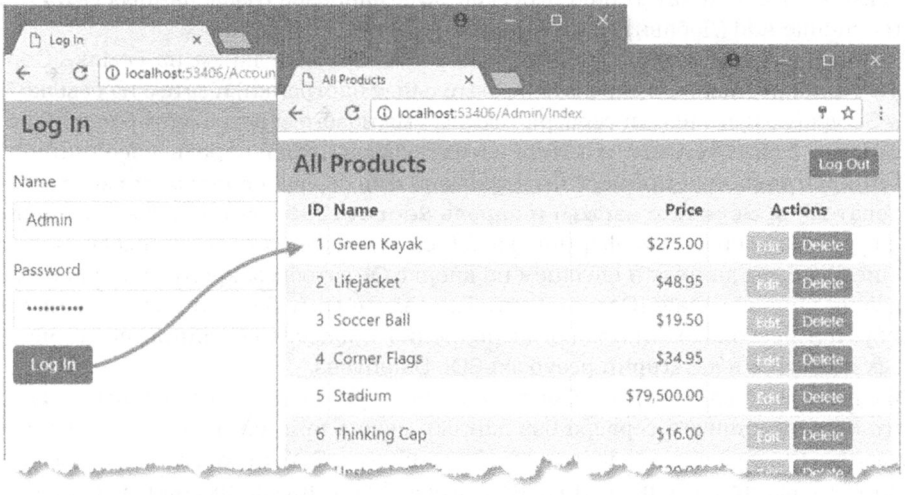

Рис. 12.1. Процесс аутентификации/авторизации для административных функций

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 применять 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 их 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 ViewResult 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.

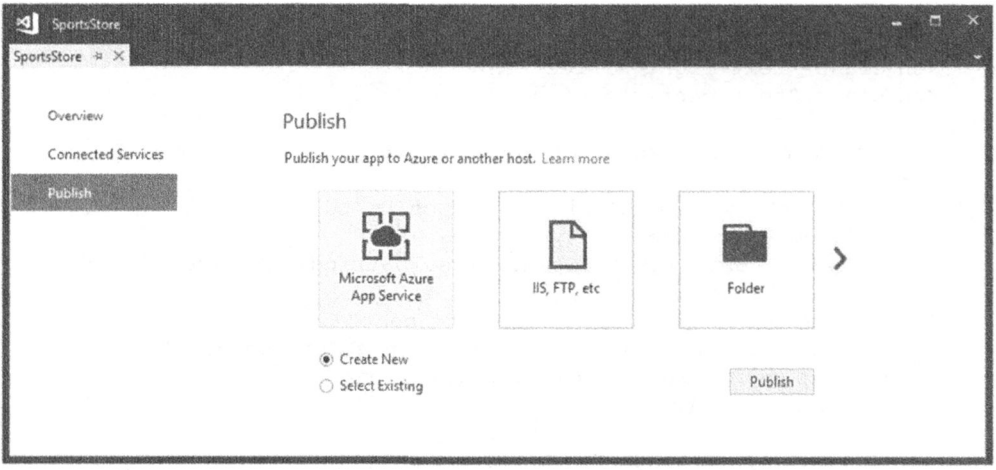

Рис. 12.2. Выбор метода опубликования

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, щелкнем 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.