

Администрирование информационных систем

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

<http://vikchas.ru>

Лабораторная работа 4_0

Тема «Миграция модели ИС и Д»

Часовских Виктор Петрович

доктор технических наук,
Профессор кафедры ШИИKM
ФГБОУ ВО «Уральский государственный
экономический университет

Екатеринбург 2023

Продолжение создания информационной системы определяет

Entity Framework Core

EF Core

#,

SQL,

SQL,

SQL

C#

EF Core

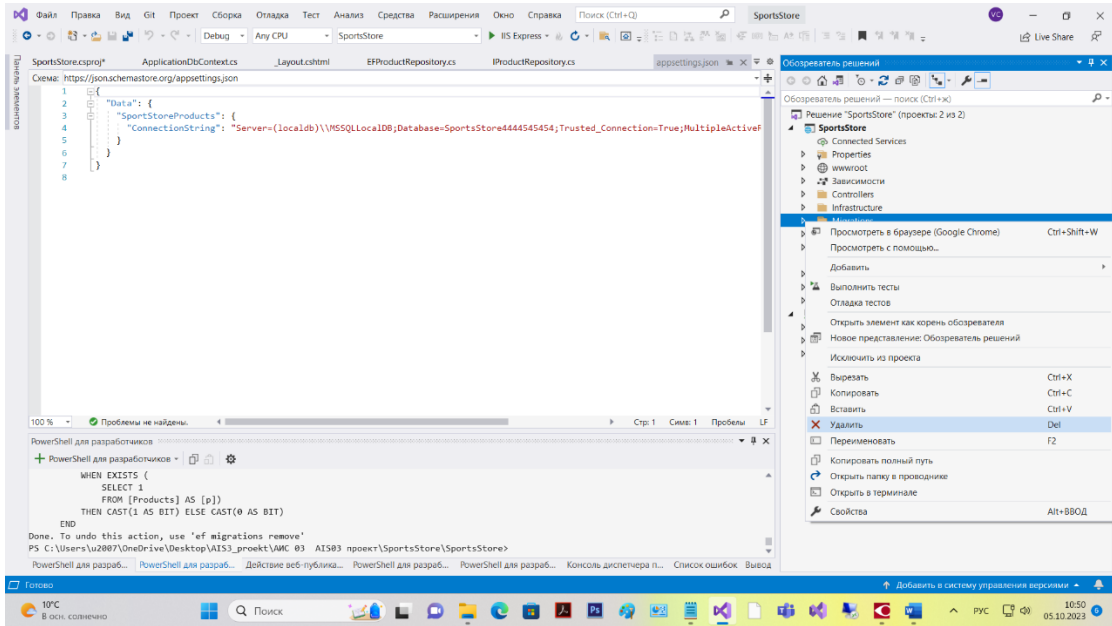
(Package

Manager Console),

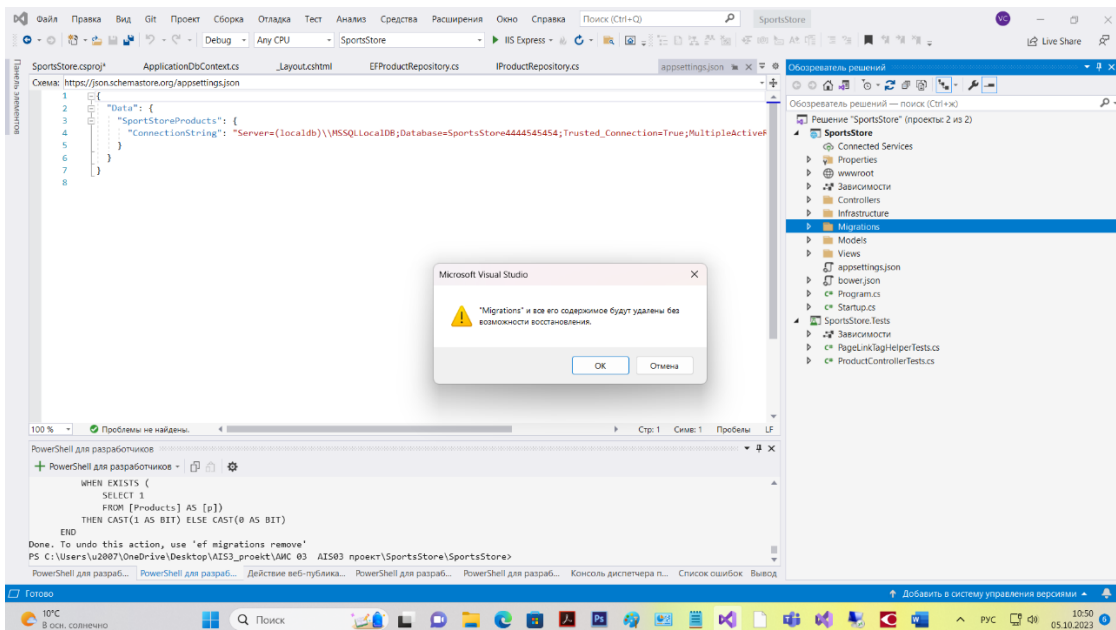
Выполнив переход от миграции к БД, необходимо построить 2-ю таблицу из

3 столбцов и создать новую БД из 2-х таблиц.

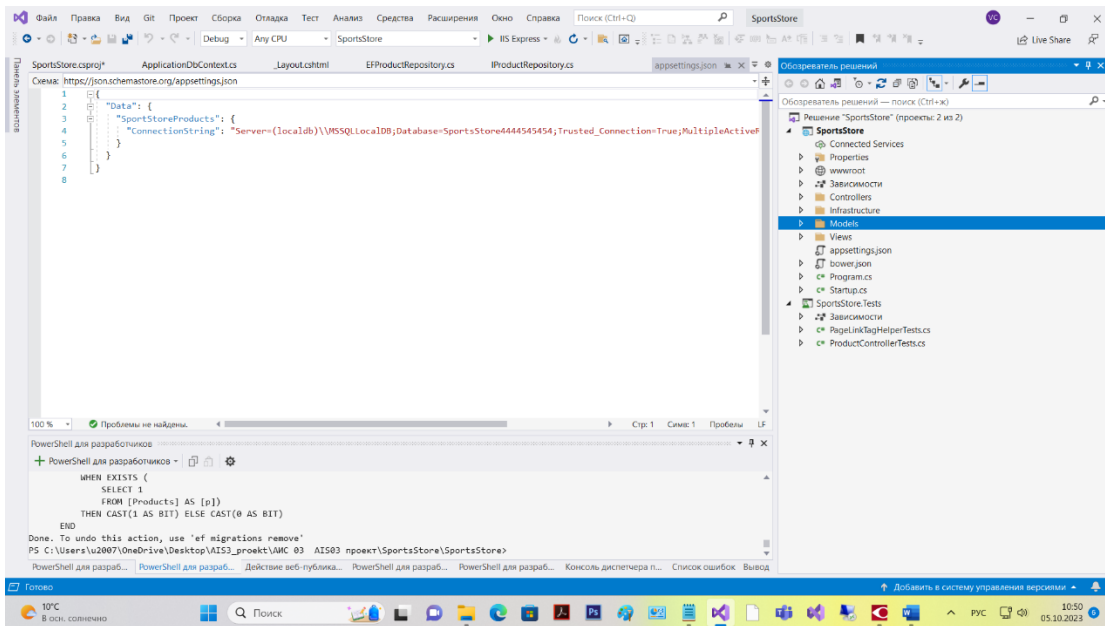
Открываем проект, находим папку Migrations



и удаляем папку Migrations

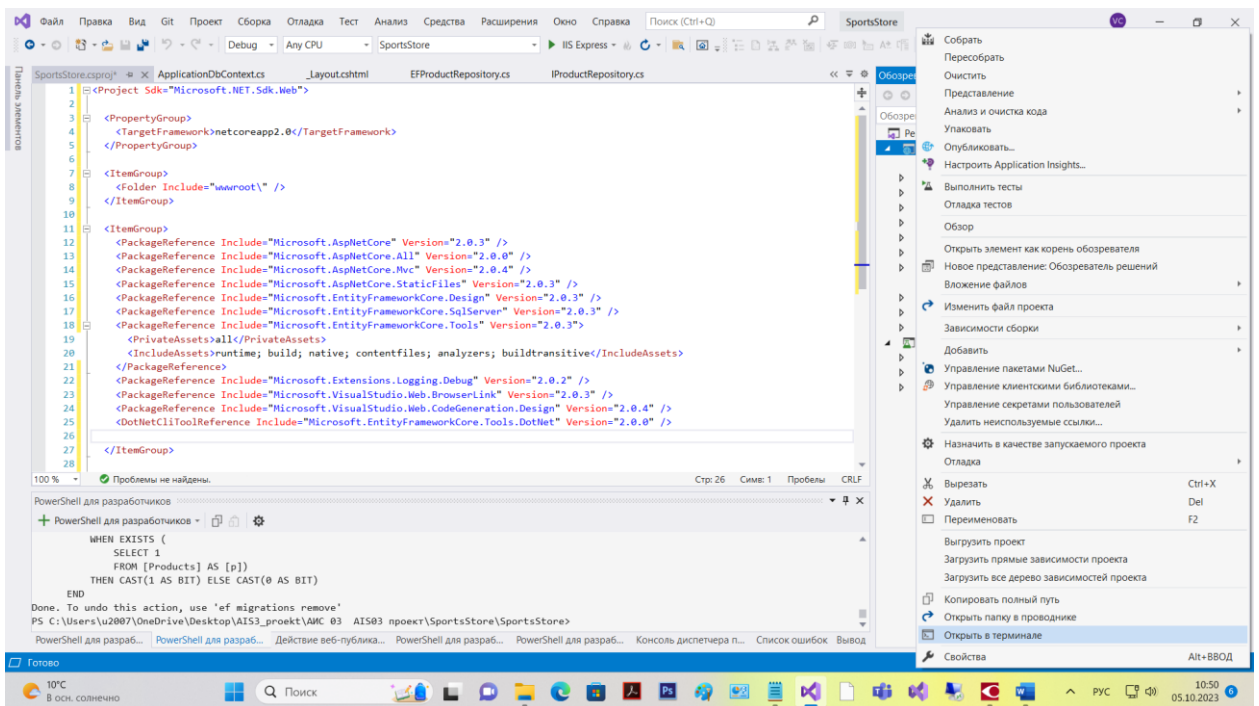


Нажимаем ОК, папка будет удалена.

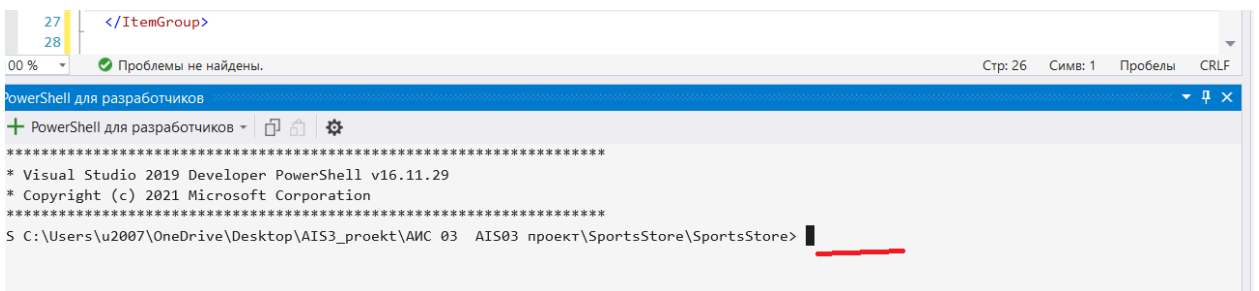


Проверяем работу миграции моделей в базу данных.

Открываем проект, курсор на название проекта, щелкаем мышку, получим список действий



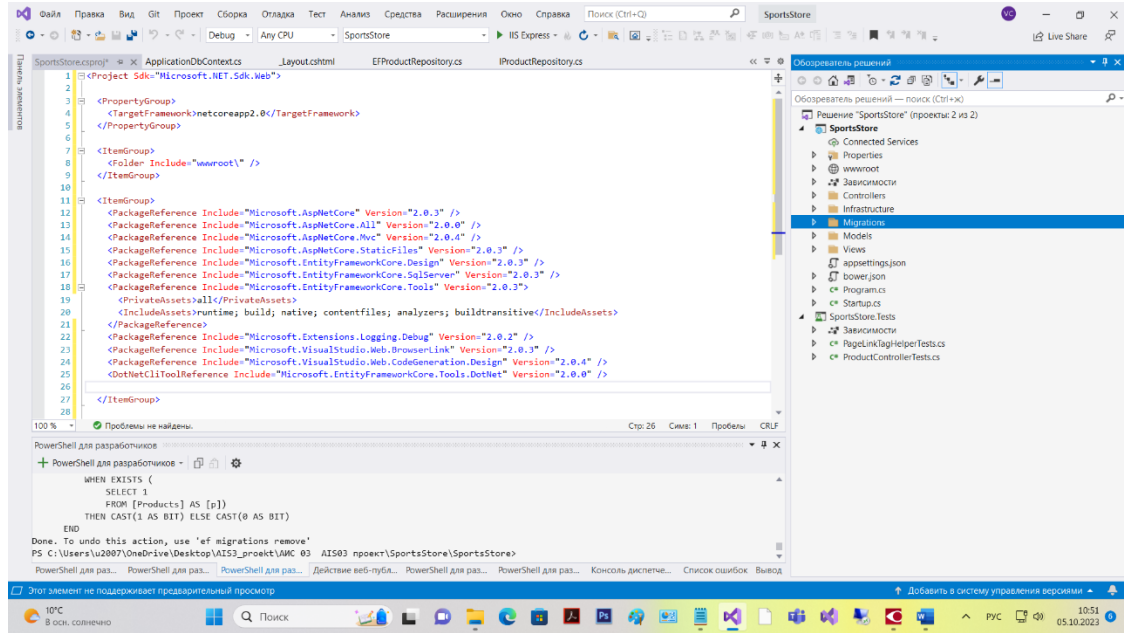
Выбираем «Открыть в терминале», получим



В поле курсора (красная линия) указываем **dotnet ef migrations add Initial**



Нажимаем Enter. Результатом будет появление папки Migrations



В проекте миграция моделей в базу данных работает.

Благодарю за внимание!



Создание классов базы данных

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

Листинг `ApplicationDbContext.cs` из папки Models

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

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



```
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options) { }
public DbSet<Product> Products { get; set; }
}
}
```

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

Создание класса хранилища

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

Листинг 1.1. Содержимое файла `EFProductRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;
namespace SportsStore.Models {
    public class EFProductRepository : IProductRepository {
        private ApplicationDbContext context;
        public EFProductRepository(ApplicationDbContext ctx) {
            context = ctx;
        }
        public IQueryable<Product> Products => context.Products;
    }
}
```

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

Определение строки подключения

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

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

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

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

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

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

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

Конфигурирование приложения

Далее необходимо прочесть строку подключения и сконфигурировать приложение для ее использования при подключении к базе данных. В листинге 8.18 показаны изменения, которые потребуются внести в класс `Startup`, чтобы получать детали данных конфигурации, содержащихся в файле `appsettings.json`, и применять их для конфигурирования `Entity Framework Core`.

Листинг Конфигурирование приложения в файле `Startup.cs` из папки `SportsStore`

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

```

using Microsoft.EntityFrameworkCore;
namespace SportsStore {
    public class Startup {
        public Startup(IConfiguration configuration) =>
            Configuration = configuration;
        public IConfiguration Configuration { get; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}

```

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

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

```

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

```

Расширяющий метод `AddDbContext()` настраивает службы, предоставляемые инфраструктурой `Entity Framework Core`, для контекста базы данных, который был создан в листинге 14.1. Как объясняется в главе 14, многие методы, применяемые в классе `Startup`, позволяют конфигурировать службы и средства промежуточного программного обеспечения с использованием аргументов `options`. Аргументом метода `AddDbContext()` является лямбда-выражение, которое получает объект `options`, конфигурирующий базу данных для класса контекста. В этом случае база данных конфигурируется с помощью метода `UseSqlServer()` и указания строки подключения, которая получена из свойства `Configuration`.

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

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

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

Отключение проверки области видимости

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

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

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

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

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

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

Создание миграции базы данных

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

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

```
dotnet ef migrations add Initial
```

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

Что насчет команд Add-Migration и Update-Database?

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

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

Создание начальных данных

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

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

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

namespace SportsStore.Models {
```

```

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

```

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

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

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

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

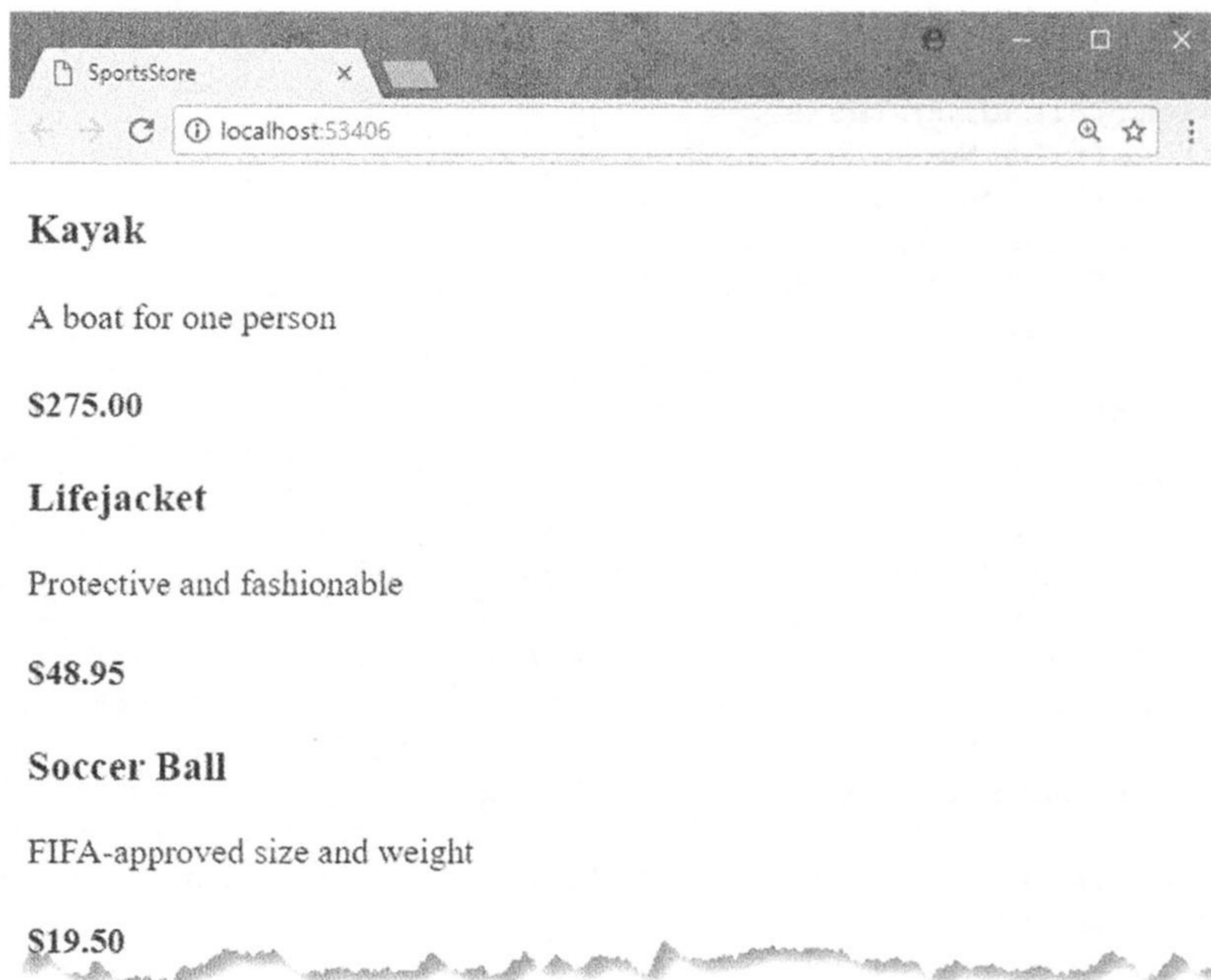
Листинг Заполнение базы данных начальной информацией в файле `Startup.cs` из папки `SportsStore`

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

namespace SportsStore {
    public class Startup {
        public Startup(IConfiguration configuration) =>
            Configuration = configuration;
        public IConfiguration Configuration { get; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

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

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



Использование хранилища в виде базы данных

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