ГЛАВА 17

Формирование шаблонов для существующих баз данных

Примеры в предшествующих главах начинались с определения классов C#, которые описывали модель и применялись для создания БД, что известно как разработка в стиле *"сначала код"*. В проектах, где необходимо использовать существующую БД, требуется другой подход, который называется разработкой в стиле *"сначала БД"*. В этой главе будет показано, как применять средство *формирования шаблонов* (scaffolding) инфраструктуры Entity Framework Core, которое инспектирует БД и автоматически генерирует модель данных. Упомянутое средство лучше всего подходит для простых БД, тогда как более сложные проекты эффективнее обслуживаются ручным моделированием данных, которое рассматривается в главе 18. В табл. 17.1 приведены сведения, позволяющие поместить формирование шаблонов в контекст.

Таблица 17.1. Помещение формирования шаблонов в контекст

Вопрос	Ответ
Что это такое?	Формирование шаблонов представляет собой процесс построения модели данных, чтобы инфраструктура Entity Framework Core могла пользоваться существующей БД
Чем оно полезно?	Не во всех проектах имеется возможность создать новую БД. Формирование шаблонов инспектирует существующую БД и автоматически создает модель данных
Как оно используется?	Формирование шаблонов выполняется с применением инструмента командной строки
Существуют ли какие-то скрытые ловушки или ограничения?	Процесс формирования шаблонов не способен иметь дело со всеми функциональными возможностями БД и может застопориться в случае крупных и сложных БД
Существуют ли альтернативы?	Вы можете моделировать БД вручную, как будет описано в главе 18

В табл. 17.2 приведена сводка по главе.

Таблица 17.2. Сводка по главе

Задача	Решение	Листинг
Формирование шаблонов для существующей БД	Запустите инструмент командной строки и подкорректируйте класс контекста для использования с инфраструктурой Entity Framework Core	17.1–17.23
Отражение в приложении изменений, внесенных в БД	Повторно сформируйте шаблоны для БД	17.24–17.30

Подготовительные шаги

Материал главы опирается на БД, которая создается без применения Entity Framework Core с целью имитации существующей БД. Для создания и заполнения БД будут использоваться средства выполнения SQL-запросов среды Visual Studio, как объясняется в последующих разделах.

На заметку! Чтобы облегчить отслеживание процесса, БД создается пошагово в нескольких листингах, приведенных далее в главе. Однако ручной набор сложных SQL-операторов чреват ошибками, поэтому лучше работать с готовым SQL-файлом из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Существующая база данных

Чтобы помочь понять SQL-код, приводимый далее в главе, стоит ознакомиться с создаваемой БД. Она будет называться `ZoomShoesDb` и хранить товары для вымышленной компании Zoom Shoes, производящей кроссовки. В табл. 17.3 описаны таблицы, которые будут добавлены в БД, а также отношения между ними. Реальные БД гораздо сложнее, чем рассматриваемый пример БД, но она обладает всеми характеристиками, необходимыми для демонстрации средств Entity Framework Core для работы с существующей БД.

Таблица 17.3. Таблицы в примере БД

Имя	Описание
`Shoes`	Это центральная таблица БД, которая содержит детальные сведения о товарах, производимых компанией. Она имеет отношения со всеми остальными таблицами
`Categories`	Эта таблица содержит набор категорий, применяемых для описания выпускаемых компанией кроссовок. Она имеет отношение “многие ко многим” с таблицей `Shoes` через таблицу `ShoeCategoryJunction`
`ShoeCategoryJunction`	Это соединяющая таблица для отношения “многие ко многим” между таблицами `Shoes` и `Categories`
`Colors`	Эта таблица содержит набор цветовых комбинаций, в которых доступны кроссовки, и имеет отношение “один ко многим” с таблицей `Shoes`
`SalesCampaigns`	Эта таблица содержит детальные сведения о кампаниях по сбыту для всех видов кроссовок и имеет отношение “один к одному” с таблицей `Shoes`

Подключение к серверу баз данных

Запустите Visual Studio, не открывая и не создавая проект. Выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) и введите `(localdb)\MSSQLLocalDB` в поле Server Name (Имя сервера). (Обратите внимание на наличие в строке одного символа \, а не двух, как требуется при определении строки подключения в файле `appsettings.json`.)

Удостоверьтесь в том, что в поле Authentication (Аутентификация) выбран вариант Windows Authentication (Аутентификация Windows), а в поле Database Name (Имя БД) — вариант <default> (<стандартное>), как показано на рис. 17.1, и щелкните на кнопке Connect (Подключиться). Среда Visual Studio откроет окно редактора запросов, в котором можно вводить и выполнять SQL-операторы.

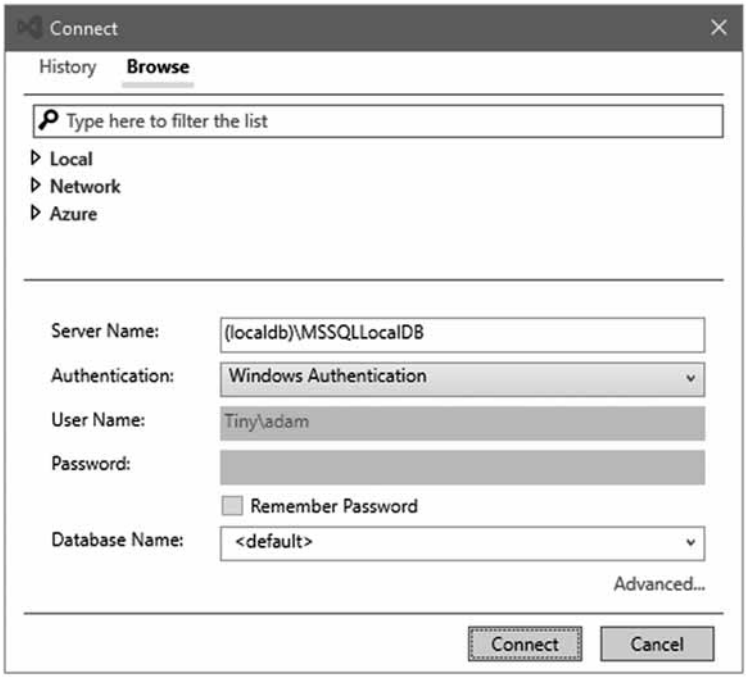

Рис. 17.1. Окно подключения к серверу баз данных

Создание базы данных

Первым делом нужно создать БД. Введите в окне редактора запросов операторы из листинга 17.1, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute (Выполнить).

На заметку! Если вы работаете со средой Visual Studio Code, тогда после щелчка правой кнопкой мыши в окне редактора запросов выберите в контекстном меню пункт Execute Query (Выполнить запрос).

Листинг 17.1. Создание БД

```
USE master

DROP DATABASE IF EXISTS ZoomShoesDb
GO

CREATE DATABASE ZoomShoesDb
GO

USE ZoomShoesDb
GO
```

Команда `DROP DATABASE` удаляет БД по имени `ZoomShoesDb`, если она уже существует, что гарантирует возможность начать сначала в случае ошибки при подготовке БД для настоящей главы. Команда `CREATE DATABASE` создает БД, а команда `USE` сообщает серверу баз данных, что следующие за ней команды должны применяться к БД по имени `ZoomShoesDb`.

Создание таблицы `Colors`

Порядок создания таблиц важен, поскольку сервер баз данных не разрешает определять отношения внешнего ключа для несуществующих таблиц. Например, таблица `Colors` должна быть создана перед таблицей `Shoes`, чтобы в таблице `Shoes` можно было определить столбец внешнего ключа, который будет использоваться для отношения "один ко многим", описанного в табл. 17.3. В окне редактора запросов введите SQL-код из листинга 17.2, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute, чтобы создать и заполнить таблицу `Colors`.

Листинг 17.2. Создание и заполнение таблицы `Colors`

```
CREATE TABLE Colors (
    Id bigint IDENTITY(1,1) NOT NULL,
    Name nvarchar(max) NOT NULL,
    MainColor nvarchar(max) NOT NULL,
    HighlightColor nvarchar(max) NOT NULL,
  CONSTRAINT PK_Colors PRIMARY KEY (Id));

SET IDENTITY_INSERT dbo.Colors ON
INSERT dbo.Colors (Id, Name, MainColor, HighlightColor)
  VALUES (1, N'Red Flash', N'Red', N'Yellow'),
         (2, N'Cool Blue', N'Dark Blue', N'Light Blue'),
         (3, N'Midnight', N'Black', N'Black'),
         (4, N'Beacon', N'Yellow', N'Green')
SET IDENTITY_INSERT dbo.Colors OFF
GO
```

Команда `CREATE TABLE` создает таблицу `Colors`, которая имеет столбцы `Id`, `Name`, `MainColor` и `HighlightColor`, причем `Id` является столбцом первичного ключа. Команда `INSERT` заполняет таблицу, а команда `SET IDENTITY` применяется для того, чтобы временно разрешить добавление данных со значениями для столбца `Id`. Таблица `Colors` сконфигурирована так, что за генерацию значений для столбца `Id` несет ответственность сервер баз данных, но при заполнении БД должны указываться значения `Id`, чтобы можно было корректно настроить отношения между таблицами.

Для проверки правильности созданной таблицы замените текст в окне редактора запросов SQL-запросом, приведенным в листинге 17.3.

Листинг 17.3. Запрашивание таблицы Colors

```
SELECT * FROM Colors
```

Щелкните правой кнопкой мыши в окне редактора запросов и выберите в контекстном меню пункт Execute; вы должны увидеть вывод, показанный в табл. 17.4, который отражает структуру и содержимое таблицы Colors.

Внимание! Если вам не удалось получить ожидаемые результаты, тогда возвратитесь к листингу 17.1 и начните сначала. У вас может возникнуть соблазн продолжить в любом случае, но вы не получите ожидаемые результаты позже в главе.

Таблица 17.4. Структура и данные таблицы Colors

Id	Name	MainColor	HighlightColor
1	Red Flash	Red (красный)	Yellow (желтый)
2	Cool Blue	Dark Blue (темно-синий)	Light Blue (светло-голубой)
3	Midnight	Black (черный)	Black (черный)

Создание таблицы Shoes

Таблица Shoes содержит подробные сведения о товарах, производимых компанией Zoom Shoes. Чтобы создать таблицу Shoes, замените текст в окне редактора запросов SQL-запросом, представленным в листинге 17.4, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute.

Листинг 17.4. Создание и заполнение таблицы Shoes

```
CREATE TABLE Shoes (
   Id bigint IDENTITY(1,1) NOT NULL,
   Name nvarchar(max) NOT NULL,
   ColorId bigint NOT NULL,
   Price decimal(18, 2) NOT NULL,
  CONSTRAINT PK_Shoes PRIMARY KEY (Id ),
  CONSTRAINT FK_Shoes_Colors FOREIGN KEY(ColorId) REFERENCES dbo.Colors (Id))
SET IDENTITY_INSERT dbo.Shoes ON
INSERT dbo.Shoes (Id, Name, ColorId, Price)
  VALUES (1, N'Road Rocket', 2, 145.0000),
         (2, N'Trail Blazer', 4, 150.0000),
         (3, N'All Terrain Monster', 3, 250.0000),
         (4, N'Track Star', 1, 120.0000)
SET IDENTITY_INSERT dbo.Shoes OFF
GO
```

Команда CREATE TABLE создает таблицу Shoes со столбцами Id, Name, ColorId и Price. Столбец Id хранит первичные ключи, а между столбцом ColorId таблицы Shoes и столбцом Id таблицы Colors определено отношение внешнего ключа.

Для проверки, корректно ли создана и заполнена таблица Shoes, замените текст в окне редактора запросов SQL-запросом, показанным в листинге 17.5.

Листинг 17.5. Запрашивание таблицы Shoes

```
SELECT * FROM Shoes
```

Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute; вы должны получить вывод, приведенный в табл. 17.5, который отражает структуру и содержимое таблицы Shoes.

Таблица 17.5. Структура и данные таблицы Shoes

Id	Name	ColorId	Price
1	Road Rocket	2	145.00
2	Trail Blazer	4	150.00
3	All Terrain Monster	3	250.00
4	Track Star	1	120.00

Создание таблицы SalesCampaigns

Таблица SalesCampaigns имеет отношение "один к одному" с таблицей Shoes и содержит детали о кампаниях по сбыту, ассоциированных со всеми товарами, где таблица Shoes — главная сущность в отношении. Чтобы создать таблицу SalesCampaigns, замените текст в окне редактора запросов SQL-запросом, представленным в листинге 17.6. Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute.

Листинг 17.6. Создание и заполнение таблицы SalesCampaigns

```
CREATE TABLE SalesCampaigns(
    Id bigint IDENTITY(1,1) NOT NULL,
    Slogan nvarchar(max) NULL,
    MaxDiscount int NULL,
    LaunchDate date NULL,
    ShoeId bigint NOT NULL,
  CONSTRAINT PK_SalesCampaigns PRIMARY KEY (Id),
  CONSTRAINT FK_SalesCampaigns_Shoes FOREIGN KEY(ShoeId)
    REFERENCES dbo.Shoes (Id),
  INDEX IX_SalesCampaigns_ShoeId UNIQUE (ShoeId))

SET IDENTITY_INSERT dbo.SalesCampaigns ON
INSERT dbo.SalesCampaigns (Id, Slogan, MaxDiscount,
  LaunchDate, ShoeId) VALUES
    (1, N'Jet-Powered Shoes for the Win!',
      20, CAST(N'2019-01-01' AS Date), 1),
    (2, N'"Blaze" a Trail with Side-Mounted Flame Throwers ',
      15, CAST(N'2019-05-03' AS Date), 2),
    (3, N'All Surfaces. All Weathers. Victory Guaranteed.',
      5, CAST(N'2020-01-01' AS Date), 3),
    (4, N'Contains an Actual Star to Dazzle Competitors',
      25, CAST(N'2020-01-01' AS Date), 4)
SET IDENTITY_INSERT dbo.SalesCampaigns OFF
GO
```

Таблица SalesCampaigns содержит столбцы Id, Slogan, MaxDiscount, LauchDate и ShoeId. Столбец Id используется для хранения первичных ключей, а столбец ShoeId является столбцом внешнего ключа, который хранит значения из столбца Id таблицы Shoes. Также имеется индекс, который требует уникальных значений в столбце ShoeId и обеспечивает отношение "один к одному" с таблицей Shoes.

Для проверки, корректно ли создана и заполнена таблица SalesCampaigns, замените текст в окне редактора запросов SQL-запросом из листинга 17.7.

Листинг 17.7. Запрашивание таблицы SalesCampaigns

```
select * from SalesCampaigns
```

Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute; вы должны получить вывод, показанный в табл. 17.6, который отражает структуру и содержимое таблицы SalesCampaigns.

Таблица 17.6. Структура и данные таблицы SalesCampaigns

Id	Slogan	MaxDiscount	LaunchDate	ShoeId
1	Jet-Powered Shoes for the Win!	20	2019-01-01	1
2	"Blaze" a Trail with Side-Mounted Flame Throwers	15	2019-05-03	2
3	All Surfaces. All Weathers. Victory Guaranteed.	5	2020-01-01	3
4	Contains an Actual Star to Dazzle Competitors	25	2020-01-01	4

Создание таблиц Categories и ShoeCategoryJunction

Для завершения БД осталось создать таблицу Categories и таблицу ShoeCategoryJunction, которая сделает возможным отношение "многие ко многим" с таблицей Shoes. Чтобы создать и заполнить указанные таблицы, замените текст в окне редактора запросов SQL-запросом, приведенным в листинге 17.8, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute.

Листинг 17.8. Создание таблиц Categories и ShoeCategoryJunction

```
CREATE TABLE Categories(
   Id bigint IDENTITY(1,1) NOT NULL,
   Name nvarchar(max) NOT NULL,
  CONSTRAINT PK_Categories PRIMARY KEY (id));

SET IDENTITY_INSERT dbo.Categories ON
INSERT dbo.Categories (Id, Name) VALUES
   (1, N'Road/Tarmac'), (2, N'Track'), (3, N'Trail'), (4, N'Road to
Trail')
SET IDENTITY_INSERT dbo.Categories OFF
GO

CREATE TABLE ShoeCategoryJunction(
   Id bigint IDENTITY(1,1) NOT NULL,
   ShoeId bigint NOT NULL,
   CategoryId bigint NOT NULL,
```

```
  CONSTRAINT PK_ShoeCategoryJunction PRIMARY KEY (Id),
  CONSTRAINT FK_ShoeCategoryJunction_Categories FOREIGN KEY(CategoryId)
    REFERENCES dbo.Categories (Id),
  CONSTRAINT FK_ShoeCategoryJunction_Shoes FOREIGN KEY(ShoeId)
    REFERENCES dbo.Shoes (Id))
SET IDENTITY_INSERT dbo.ShoeCategoryJunction ON
INSERT dbo.ShoeCategoryJunction (Id, ShoeId, CategoryId)
VALUES (1, 1, 1), (2, 2, 3), (3, 2, 4), (4, 3, 1),
       (5, 3, 2), (6, 3, 3), (7, 3, 4), (8, 4, 2)
SET IDENTITY_INSERT dbo.ShoeCategoryJunction OFF
GO
```

Таблица `Categories` имеет столбцы `Id` и `Name`, причем столбец `Id` применяется для хранения первичных ключей. Таблица `ShoeCategoryJunction` содержит столбцы `Id`, `ShoeId` и `CategoryId`; столбец `Id` используется для хранения первичных ключей, а остальные столбцы применяются для отношений внешнего ключа с таблицами `Shoes` и `Categories`. Здесь задействован тот же подход к поддержке отношения "многие ко многим", который был описан в главе 16.

Для проверки, корректно ли созданы и заполнены таблицы `Categories` и `ShoeCategoryJunction`, замените текст в окне редактора запросов SQL-запросом из листинга 17.9.

Листинг 17.9. Запрашивание таблиц `Categories` и `ShoeCategoryJunction`

```
SELECT * FROM Categories
INNER JOIN ShoeCategoryJunction
ON Categories.Id = ShoeCategoryJunction.ShoeId
```

Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute; вы должны получить вывод, показанный в табл. 17.6, который отражает структуру и содержимое двух таблиц.

Таблица 17.7. Структура и данные таблиц `Categories` и `ShoeCategoryJunction`

Id	Name	Id	ShoeId	CategoryId
1	Road/Tarmac	1	1	1
2	Track	2	2	3
2	Track	3	2	4
3	Trail	4	3	1
3	Trail	5	3	2
3	Trail	6	3	3
3	Trail	7	3	4
4	Road to Trail	8	4	2

Создание проекта ASP.NET Core MVC

В дополнение к БД необходим проект ASP.NET Core MVC, чтобы можно было продемонстрировать использование инфраструктуры Entity Framework Core с существующей БД. Для создания проекта выберите в меню File (Файл) среды Visual Studio пункт

New⇨Project (Создать⇨Проект), укажите шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core) и введите `ExistingDb` в поле Name (Имя), как показано на рис. 17.2. (Может отобразиться предложение сохранить содержимое окна редактора запросов. Если вы получили ожидаемые результаты для всех запросов, то введенные ранее SQL-операторы больше не нужны.) Щелкните на кнопке OK, чтобы инициировать процесс создания проекта.

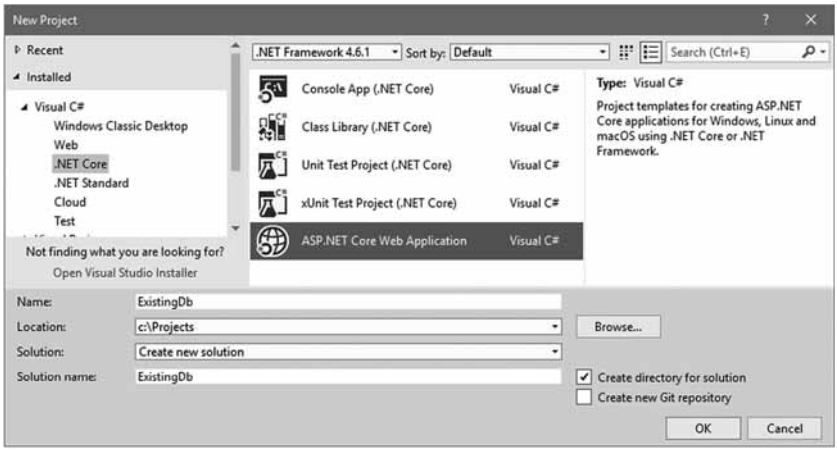

Рис. 17.2. Создание нового проекта ASP.NET Core MVC

Удостоверьтесь, что в списках в левой верхней части окна выбраны варианты .NET Core и ASP.NET Core 2.0, и щелкните на шаблоне Empty (Пустой), как показано на рис. 17.3. Щелкните на кнопке OK, чтобы завершить процесс конфигурирования и создать новый проект.

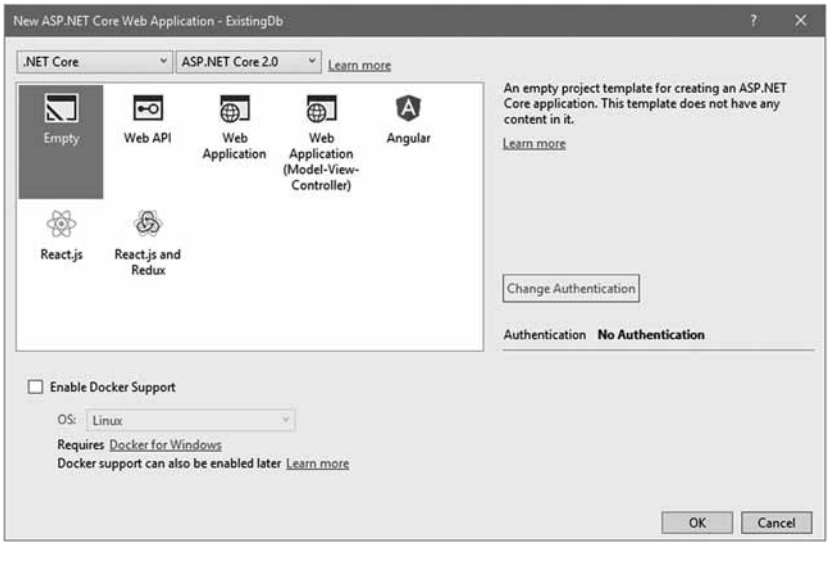

Рис. 17.3. Конфигурирование нового проекта ASP.NET Core MVC

Добавление NuGet-пакета Tools

Мета-пакет, который среда Visual Studio добавляет в новые проекты, содержит функциональность ASP.NET Core MVC и Entity Framework Core, но требуется еще одно ручное добавление для установки инструментов командной строки Entity Framework Core. Щелкните правой кнопкой мыши на элементе проекта ExistingDb в окне Solution Explorer, выберите в контекстном меню пункт Edit ExistingDb.csproj (Редактировать ExistingDb.csproj) и добавьте элемент конфигурации из листинга 17.10.

Листинг 17.10. Добавление пакета Tools в файле ExistingDb.csproj из папки ExistingDb

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

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

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.5"
/>
    <DotNetCliToolReference
      Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="2.0.0" />
    </ItemGroup>

</Project>
```

Сохраните файл, в результате чего пакет будет загружен и установлен, снабдив проект инструментами командной строки, которые нужны для проработки примеров в этой главе.

Конфигурирование служб и промежуточного программного обеспечения ASP.NET Core

Шаблон Empty требует конфигурирования для настройки промежуточного ПО и служб в классе Startup, необходимом приложению MVC (листинг 17.11).

Листинг 17.11. Конфигурирование промежуточного ПО и служб в файле Startup.cs из папки ExistingDb

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

namespace ExistingDb {
  public class Startup {
```

```
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Операторы конфигурации для Entity Framework Core пока не добавлялись. Службы, требующиеся для работы с БД, будут настроены позже в главе.

Добавление контроллера и представления и установка Bootstrap

Финальный подготовительный шаг касается создания простого контроллера и представления, чтобы приложение можно было протестировать, и установка пакета Bootstrap CSS, предназначенного для стилизации отображаемого пользователю HTML-содержимого. Создайте папку `Controllers` и добавьте в нее файл класса по имени `HomeController.cs` с содержимым, приведенным в листинге 17.12.

Листинг 17.12. Содержимое файла `HomeController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;

namespace ExistingDb.Controllers {

  public class HomeController : Controller {

    public IActionResult Index() {
      return View();
    }
  }
}
```

Контроллер будет строиться позже в главе, а пока в нем есть один метод действия, который выбирает стандартное представление. Чтобы создать представление, добавьте папку `Views/Home` и поместите в нее файл по имени `Index.cshtml` с содержимым из листинга 17.13.

Листинг 17.13. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@{
  ViewData["Title"] = "Existing Database";
  Layout = "_Layout";
}

<h2 class="bg-info p-1 m-1 text-white">Placeholder for Data</h2>
```

Чтобы снабдить представления в примере приложения разделяемой компоновкой, создайте папку `Views/Shared` и добавьте в нее файл по имени `_Layout.cshtml`, содержимое которого показано в листинге 17.14.

Листинг 17.14. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewData["Title"]</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.
css" />
</head>
<body>
  <div class="p-2">
    @RenderBody()
  </div>
</body>
</html>
```

Для установки пакета Bootstrap CSS воспользуйтесь шаблоном элемента JSON File (Файл JSON), находящимся в категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), и создайте файл по имени `.bowerrc` с содержимым из листинга 17.15. (Важно обратить внимание на имя файла: оно начинается с точки, содержит две буквы `r` и не имеет расширения.)

Листинг 17.15. Содержимое файла `.bowerrc` из папки `ExistingDb`

```
{
  "directory": "wwwroot/lib"
}
```

Снова воспользуйтесь шаблоном элемента JSON File и создайте файл по имени `bower.json` с содержимым из листинга 17.16.

Листинг 17.16. Содержимое файла `bower.json` из папки `ExistingDb`

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

После сохранения изменений в файле пакет Bootstrap будет загружен и установлен в папку `wwwroot/lib`.

Тестирование примера приложения

Для упрощения работы с приложением отредактируйте файл `Properties/launchSettings.json`, изменив содержащиеся в нем URL, чтобы они оба указывали на порт `5000` (листинг 17.17). Данный порт будет применяться в URL для демонстрации функциональных возможностей примера приложения.

Листинг 17.17. Изменение портов в файле `launchSettings.json` из папки `Properties`

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "ExistingDb": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Запустите приложение, используя команду `dotnet run` в папке проекта `ExistingDb`, и перейдите в браузере по ссылке `http://localhost:5000`, чтобы увидеть содержимое-заполнитель (рис. 17.4).

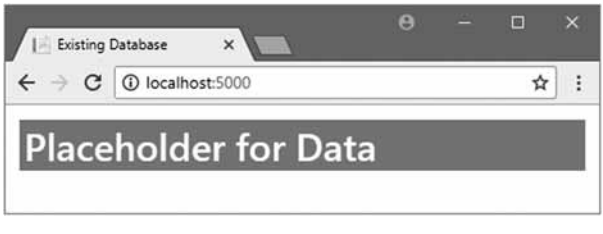

Рис. 17.4. Выполнение примера приложения

Формирование шаблонов для существующей базы данных

Самый легкий способ работы с существующей БД предусматривает применение средства формирования шаблонов Entity Framework Core, которое инспектирует БД и создает классы контекста и модели, требуемые для выполнения запросов и других операций над данными.

Даже если вы создаете модель данных для существующей БД вручную, как будет описано в главе 18, средство формирования шаблонов все равно полезно, поскольку оно позволяет удостовериться в корректности получившегося описания БД. В последующих разделах объясняется, как использовать средство формирования шаблонов и как применять созданную им модель данных в приложении ASP.NET Core MVC.

Выполнение процесса формирования шаблонов

Процесс формирования шаблонов выполняется с использованием инструмента командной строки, содержащегося в пакете, который был добавлен к проекту в листинге 17.10. Выполните в папке проекта `ExistingDb` команду из листинга 17.18 для инспектирования имеющейся БД и генерации модели данных.

Совет. Печатная страница затрудняет представление сложных команд, и вы должны позаботиться о вводе всех частей команды, показанной в листинге 17.18, в единственной строке.

Листинг 17.18. Выполнение формирования шаблонов для существующей БД

```
dotnet ef dbcontext scaffold
"Server=(localdb)\MSSQLLocalDB;Database=ZoomShoesDb"
"Microsoft.EntityFrameworkCore.SqlServer"
--output-dir "Models/Scaffold"
--context ScaffoldContext
```

Основная команда `dotnet ef dbcontext scaffold` запускает процесс формирования шаблонов. Первые два аргумента являются обязательными и предоставляют строку подключения для моделируемой БД и имя поставщика БД. В листинге 17.18 присутствуют два необязательных аргумента: аргумент `--output-dir` указывает каталог (и пространство имен) для классов C#, генерируемых процессом формирования шаблонов, а аргумент `--context` указывает имя класса контекста, применяемого для доступа к БД.

Работа с проблемными базами данных

Процесс формирования шаблонов как результат выполнения команды в листинге 17.18 проходит гладко, поскольку БД в рассматриваемом примере проста и создана специально для целей главы. При формировании шаблонов для реальной БД важно внимательно следить за выводом из команды `dotnet ef`, потому что именно в нем сообщается о любых возникших проблемах.

Одни проблемы порождают простые предупреждения, когда определенный аспект БД не может быть четко отображен на инфраструктуру Entity Fframework Core, приводя в итоге к компромиссу в созданной модели данных. Но могут возникать также и накладки, если инфраструктура Entity Fframework Core не знает, каким образом продолжать. В таких ситуациях вы можете либо моделировать БД вручную, как объясняется в главе 18, либо ограничить область действия процесса формирования шаблонов, используя аргумент `--table`, чтобы выбрать таблицы, с которыми нужно работать, и исключить те, которые вызывают проблемы.

В листинге 17.18 указана строка подключения для БД, созданной в начале главы, и выбран поставщик SQL Server. Необязательные аргументы указывают, что классы модели данных должны быть помещены в папку `Models/Scaffold`, а классу контекста необходимо назначить имя `ScaffoldContext`.

Выполнение команды занимает некоторое время. Когда она завершится, вы увидите, что в окне Solution Explorer появилась папка `Models/Scaffold`, содержащая классы, которые представляют таблицы в БД, а также класс контекста `ScaffoldContext`.

Каждый класс, сгенерированный процессом формирования шаблонов, следует приблизительно такому же стилю, который был описан в предшествующих главах. Скажем, вот содержимое файла `Shoes.cs`, который будет применяться для представления строк данных из таблицы `Shoes` в БД:

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

namespace ExistingDb.Models.Scaffold {

  public partial class Shoes {

    public Shoes() {
      ShoeCategoryJunction = new HashSet<ShoeCategoryJunction>();
    }

    public long Id { get; set; }
    public string Name { get; set; }
    public long ColorId { get; set; }
    public decimal Price { get; set; }

    public Colors Color { get; set; }
    public SalesCampaigns SalesCampaigns { get; set; }
    public ICollection<ShoeCategoryJunction>
      ShoeCategoryJunction { get; set; }
  }
}
```

Легко заметить, что этот класс очень похож на классы, созданные в предшествующих главах. Свойства `Id`, `Name`, `ColorId` и `Price` соответствуют столбцам, добавленным в таблицу `Shoes`. Свойства `Color`, `SalesCampaigns` и `ShoeCategoryJunction` делают возможным переход на связанные данные, соответствуя отношениям между таблицами в БД.

Существует ряд мелких отличий. Например, именем класса является `Shoes`, т.к. процесс формирования шаблонов использует для него имя таблицы БД. Кроме того, конструктор инициализирует коллекцию для свойства `ShoeCategoryJunction`, что я опускал, предпочитая иметь дело со значениями `null`, а не с пустой коллекцией, когда доступные данные отсутствуют.

Процесс формирования шаблонов также создает класс контекста, в котором предусмотрены свойства `DbSet<T>` для всех таблиц в БД. Вдобавок класс контекста переопределяет методы `OnConfiguring()` и `OnModelCreating()`, которые ранее не требовались. Ниже приведено содержимое файла `ScaffoldContext.cs`, где ради краткости ряд операторов из метода `OnModelCreating` не показан:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;
```

```
namespace ExistingDb.Models.Scaffold {
  public partial class ScaffoldContext : DbContext {
    public virtual DbSet<Categories> Categories { get; set; }
    public virtual DbSet<Colors> Colors { get; set; }
    public virtual DbSet<SalesCampaigns> SalesCampaigns { get; set; }
    public virtual DbSet<ShoeCategoryJunction>
      ShoeCategoryJunction { get; set; }
    public virtual DbSet<Shoes> Shoes { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder) {
      if (!optionsBuilder.IsConfigured) {
        optionsBuilder.UseSqlServer(
          @"Server=(localdb)\MSSQLLocalDB;Database=ZoomShoesDb");
      }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      // ...для краткости остальные операторы не показаны...

      modelBuilder.Entity<Shoes>(entity => {
        entity.Property(e => e.Name).IsRequired();

        entity.HasOne(d => d.Color)
          .WithMany(p => p.Shoes)
          .HasForeignKey(d => d.ColorId)
          .OnDelete(DeleteBehavior.ClientSetNull)
          .HasConstraintName("FK_Shoes_Colors");
      });
    }
  }
}
```

Функциональность, обеспечиваемая кодом в этих методах, в предшествующих главах находилась в других местах. Метод `OnConfiguring()` содержит строку подключения для БД, которая определяется в файле настроек проекта ASP.NET Core MVC. Метод `OnModelCreating()` содержит операторы, которые создают отображение между БД и классами модели данных, и они были частью класса снимка, создаваемого миграцией. Строка подключения вскоре будет перенесена в обычное место, а назначение операторов в методе `OnModelCreating()` объясняется в главе 18.

Использование модели данных, сгенерированной процессом формирования шаблонов, в ASP.NET Core MVC

После создания модель данных можно применять в ASP.NET Core MVC, внеся лишь несколько изменений, которые описаны в последующих разделах.

Создание файла конфигурации

Первый шаг — создание файла конфигурации, содержащего строку подключения, которую процесс формирования шаблонов поместил в класс контекста (что не подходит для приложений ASP.NET Core MVC). Воспользуйтесь шаблоном элемента ASP.NET Configuration File (Файл конфигурации ASP.NET), чтобы добавить в проект файл по имени `appsettings.json` и внести в него изменения, показанные в листинге 17.19.

Листинг 17.19. Содержимое файла appsettings.json из папки ExistingDb

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=ZoomShoesDb;
MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "None",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  }
}
```

В дополнение к изменению строки подключения в файл appsettings.json добавлен раздел конфигурации Logging, который заставит инфраструктуру Entity Framework Core отображать детали запросов, отправляемых серверу баз данных. Изменения, касающиеся ведения журналов, не требуются для работы модели данных, но полезны, поскольку позволяют видеть, каким образом инфраструктура Entity Framework Core взаимодействует с БД.

Обновление класса контекста

Класс контекста, созданный процессом формирования шаблонов, должен быть модифицирован, прежде чем его можно будет применять с остальными частями приложения ASP.NET Core MVC. Удалите или закомментируйте код метода OnConfiguring() и замените его конструктором, который принимает параметры конфигурации через внедрение зависимостей (листинг 17.20).

Листинг 17.20. Обновление класса контекста в файле ScaffoldContext.cs из папки Models/Scaffold

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

namespace ExistingDb.Models.Scaffold {

  public partial class ScaffoldContext : DbContext {

    public virtual DbSet<Categories> Categories { get; set; }
    public virtual DbSet<Colors> Colors { get; set; }
    public virtual DbSet<SalesCampaigns> SalesCampaigns { get; set; }
    public virtual DbSet<ShoeCategoryJunction>
      ShoeCategoryJunction { get; set; }
    public virtual DbSet<Shoes> Shoes { get; set; }

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

    // protected override void OnConfiguring(DbContextOptionsBuilder
    //    optionsBuilder) {
    //   if (!optionsBuilder.IsConfigured) {
    //    optionsBuilder.UseSqlServer
    //      (@"Server=(localdb)\MSSQLLocalDB;Database=ZoomShoesDb");
```

```
    //   }
    // }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      // ...для краткости операторы не показаны...
    }
  }
}
```

Регистрация промежуточного программного обеспечения и служб

Процесс формирования шаблонов не конфигурирует инфраструктуру Entity Framework Core для использования с ASP.NET Core MVC и не регистрирует класс контекста как службу для внедрения зависимостей. Чтобы обеспечить совместную работу разных частей приложения, добавьте в класс `Startup` операторы, выделенные полужирным в листинге 17.21.

Листинг 17.21. Конфигурирование приложения в файле `Startup.cs` из папки `ExistingDb`

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

namespace ExistingDb {
  public class Startup {
    public Startup(IConfiguration config) => Configuration = config;
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<ScaffoldContext>(options =>
        options.UseSqlServer(conString));
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```