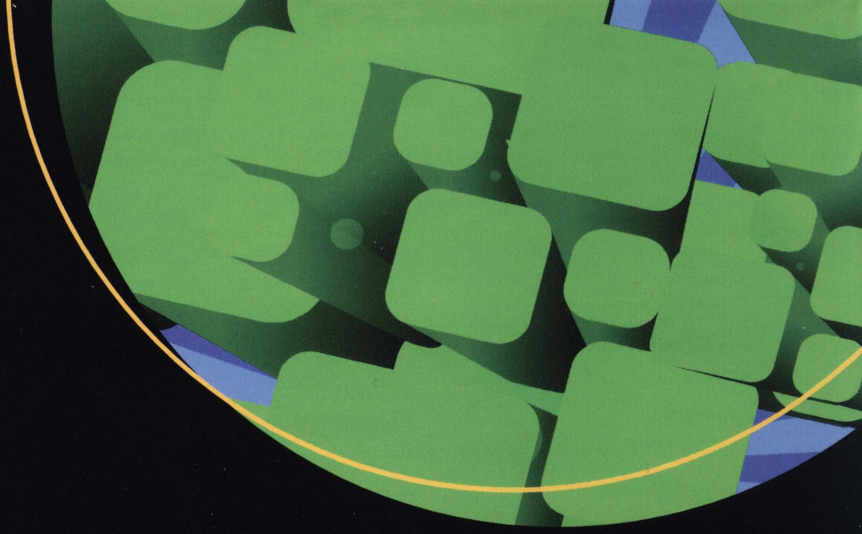

Entity Framework Core 2 для ASP.NET Core MVC для профессионалов

Адам Фримен

Диалектика
www.williamspublishing.com
Apress®

Entity Framework Core 2 для ASP.NET Core MVC

для профессионалов

Pro Entity Framework Core 2 for ASP.NET Core MVC

Adam Freeman

Apress®

Entity Framework Core 2 для ASP.NET Core MVC

для профессионалов

Адам Фримен

Диалектика

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Ф88
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фримен, Адам.

Ф88 Entity Framework Core 2 для ASP.NET Core MVC для профессионалов. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 624 с. : ил. — Парал. тит. англ.

ISBN 978-5-907114-86-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Адам Фримен

Entity Framework Core 2 для ASP.NET Core MVC для профессионалов

Подписано в печать 21.01.2019. Формат 70×100/16
Гарнитура Times
Усл. печ. л. 50,31. Уч.-изд. л. 35,4
Тираж 400 экз. Заказ № 966.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907114-86-9 (рус.)
ISBN 978-1-4842-3434-1 (англ.)

Оглавление

Содержание

Посвящается моей прекрасной жене Джеки Гриффитс.

Об авторе

Адам Фримен — опытный специалист в области информационных технологий, занимавший ведущие позиции во многих компаниях, последней из которых был глобальный банк, где он работал на должностях директора по внедрению технологий и руководителя административной службы. После ухода из банка Адам уделяет все свое время писательской деятельности и бегу на длинные дистанции.

О техническом рецензенте

Фабио Клаудио Ферраччати — ведущий консультант и главный аналитик/разработчик, использующий технологии Microsoft. Он работает в компании BluArancio (`www.bluarancio.com`). Фабио является сертифицированным Microsoft разработчиком решений для .NET (Microsoft Certified Solution Developer for .NET), сертифицированным Microsoft разработчиком приложений для .NET (Microsoft Certified Application Developer for .NET), сертифицированным Microsoft профессионалом (Microsoft Certified Professional), а также продуктивным автором и техническим рецензентом. За последние десять лет он написал множество статей для итальянских и международных журналов и выступал в качестве соавтора в более чем 10 книгах по разнообразным темам, связанным с компьютерами.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

ЧАСТЬ I

Введение в инфраструктуру Entity Framework Core 2

Первая часть книги поможет понять в общих чертах основополагающие идеи разработки приложений с использованием Entity Framework Core 2 и увидеть на деле, как применяется инфраструктура в проекте ASP.NET Core MVC.

ГЛАВА 1

Основы Entity Framework Core

Инфраструктура Entity Framework Core (также известная как EF Core) представляет собой пакет объектно-реляционного отображения (object-relational mapping — ORM) производства Microsoft, который позволяет приложениям .NET Core хранить данные в реляционных базах данных.

Понятие Entity Framework Core

Инфраструктура Entity Framework Core решает одну основную задачу: сохранение объектов .NET в базе данных (БД) и извлечение их в более позднее время. Иными словами, Entity Framework Core действует в качестве связки между приложением ASP.NET Core MVC и БД, как показано на рис. 1.1.

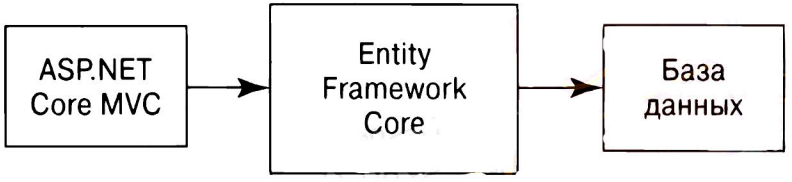

Рис. 1.1. Место Entity Framework Core

Сохранение объектов .NET в БД может оказаться неожиданно сложным процессом. Базы данных не существуют в изоляции. Они создаются и управляются *серверами баз данных*, которые являются специализированными приложениями, ориентированными исключительно на хранение и управление данными. Серверы баз данных предоставляют постоянное хранилище для большинства приложений, а самые популярные серверы баз данных разрабатывались десятилетиями, результатом чего стало высокопроизводительное и надежное программное обеспечение, битком набитое функциональными средствами. Серверы баз данных поддерживают основной набор общих средств, но сами они различаются за счет специальных дополнений, и достижение наилучших результатов означает использование в своих интересах этих специальных дополнений. Доступны различные типы серверов баз данных; вид, с которым работает Entity Framework Core, называется *сервером реляционных баз данных*, также известным как *система управления реляционными базами данных* (СУРБД). Сервер реля-

ционных баз данных управляет *реляционными базами данных*, где данные хранятся как строки в таблицах, что довольно похоже на устройство электронной таблицы. Серверы реляционных баз данных обычно принимают команды, выраженные на *языке структурированных запросов* (Structured Query Language — SQL), который позволяет представлять операции с данными, такие как сохранение или удаление данных. Существует стандарт SQL, но серверы баз данных применяют слегка отличающиеся диалекты, особенно когда дело доходит до обращения к нестандартным средствам.

На заметку! Ради краткости описание здесь упрощено. Мир баз данных переполнен отличительными признаками между разными типами баз данных и серверов баз данных, ни один из которых не имеет большого значения при использовании ASP.NET Core MVC или Entity Framework Core.

Для сохранения объектов .NET в БД инфраструктура Entity Framework Core должна быть в состоянии преобразовать объекты в форму, допускающую хранение в таблице БД, и выразить SQL-команду, которую может обработать сервер баз данных. Чтобы справиться с различиями между серверами баз данных, инфраструктура Entity Framework Core полагается на *поставщик базы данных*, который отвечает за взаимодействие с БД и формулирование SQL-команд. Для извлечения объектов .NET из БД инфраструктура Entity Framework Core должна уметь выполнять обратный процесс. Она обязана создать SQL-команду, которая запросит у сервера баз данных значения данных, представляющие объект, и применит их при заполнении свойств объекта .NET. Чтобы сделать такой процесс как можно более естественным, для запрашивания БД в Entity Framework Core поддерживается язык LINQ, который делает работу с коллекциями объектов, хранящихся в БД, похожей на работу с коллекциями объектов в памяти.

Об этой книге

В книге объясняется, как использовать Entity Framework Core в приложениях ASP.NET Core MVC. В ней будут показаны различные линии поведения, которые Entity Framework способна добавлять в проект, и ловушки, куда по неосторожности можно попасть. Инфраструктура Entity Framework Core — мощный инструмент, но важно уделять внимание деталям, иначе в итоге получится приложение, которое не функционирует надлежащим образом или не ведет себя так, как ожидалось.

Что необходимо знать?

Чтобы извлечь из книги максимальную пользу, вы должны быть знакомы с разработкой приложений ASP.NET Core MVC. Вы будете испытывать трудности, если не знаете, как работают контроллеры и действия, или не понимаете, каким образом ведут себя представления Razor. Если вы не знакомы с ASP.NET Core MVC, тогда почитайте документацию по ссылке `https://docs.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-2.1` или книгу *ASP.NET Core MVC 2 с примерами на C# для профессионалов, 7-е изд.* ("Диалектика", 2019 год).

Какое программное обеспечение потребуется?

Для проработки примеров, приводимых в книге, необходимо иметь компьютер с системой Windows, комплектом .NET Core SDK и последней версией среды Visual Studio. За исключением Windows все инструменты, применяемые в книге, доступны бесплатно, а в главе 2 объясняется, как настроить среду разработки.

Что, если вы не хотите использовать Windows?

Повсюду в книге материал основан на системе Windows и среде Visual Studio, поскольку они применяются большинством читателей, а возможность использования LocalDB (версии продукта SQL Server для разработчиков, доступной только для Windows) делает примеры более предсказуемыми и надежными. Приложив небольшие усилия, можно обеспечить запуск всех примеров на любой платформе, поддерживающей .NET Core, хотя понадобится установить полный продукт SQL Server (или применять контейнеры Docker).

Какова структура книги?

Книга разделена на три части, в каждой из которых раскрывается набор связанных тем.

Часть I. Введение в инфраструктуру Entity Framework Core 2

Учиться лучше всего, что-то делая, и в этой части книги вы получите высокоуровневое представление о том, каким образом работает инфраструктура Entity Framework Core и как она интегрируется с ASP.NET Core MVC. В главе 2 вы создадите свое первое приложение ASP.NET Core MVC, которое использует Entity Framework Core для сохранения данных. В главе 3 предлагается учебник для работы с базами данных и применения языка SQL, так что вы сумеете понять, каким образом функционирует Entity Framework Core, и прорабатывать примеры по всей книге. В главах 4–10 мы займемся разработкой проекта по имени SportsStore, с помощью которого будет продемонстрирован реалистичный процесс разработки, затрагивающий наиболее важные средства Entity Framework Core, и указано, где в книге каждое средство описывается более подробно.

Часть II. Подробные сведения об инфраструктуре Entity Framework Core 2

В части II рассматриваются основные средства Entity Framework Core, которые вы будете использовать повседневно в своих проектах ASP.NET Core MVC. Вы узнаете, как работает каждое средство, поймете исполняемую им роль и освоите альтернативные приемы, когда они доступны.

Часть III. Расширенные возможности Entity Framework Core 2

В части III описаны расширенные возможности, которые предоставляет инфраструктура Entity Framework Core. Такие средства вряд ли будут применяться часто, но они бесценны, когда стандартное поведение Entity Framework Core не решает имеющейся задачи.

Где можно получить код примеров?

Примеры проектов для всех глав книги доступны для загрузки на веб-сайте издательства и по ссылке:

```
https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc
```

Загруженный файл содержит все классы, представления и другие ресурсы, позволяющие прорабатывать примеры, не вводя программный код полностью.

Резюме

В главе была представлена инфраструктура Entity Framework Core, объяснена ее роль и описаны структура и содержимое всей книги. В следующей главе ASP.NET Core MVC и Entity Framework Core будут показаны в действии на простой демонстрации совместной работы этих двух мощных инструментов.

ГЛАВА 2

Ваше первое приложение Entity Framework Core

Лучший способ начать работу с инфраструктурой Entity Framework Core — сразу же приступить к ее использованию. В настоящей главе с применением Entity Framework Core и ASP.NET Core MVC создается простое приложение, чтобы вы могли увидеть, как все совмещается друг с другом. Ради упрощения примера некоторые детали, описываемые в последующих главах, здесь опущены.

Подготовка

Для подготовки к реализации примера в этой главе — и примеров в будущих главах — вам придется установить несколько инструментов разработки. Все инструменты, требующиеся при разработке с использованием ASP.NET Core MVC и Entity Framework Core, предлагают бесплатные версии, которые мы и будем применять повсеместно в книге.

Обновления для этой книги

В Microsoft приняли активный график разработки для .NET Core, ASP.NET Core MVC и Entity Framework Core, т.е. ко времени чтения вами книги могут быть доступны их новые выпуски. Вам вряд ли придется покупать новую книгу каждые несколько месяцев, особенно с учетом того, что большинство изменений будут относительно небольшими.

Взамен в хранилище GitHub для книги (`https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`) будут помещаться критические изменения, вызванные младшими выпусками.

Такой вид обновления носит для меня (и для Apress) экспериментальный характер, и пока еще не вполне ясно, какую форму могут принять изменения (не в последнюю очередь потому, что содержимое будущих крупных выпусков ASP.NET Core MVC или Entity Framework Core не известно), но целью является продление времени жизни книги за счет дополнения рассмотренных в ней примеров.

Я не даю никаких обещаний относительно того, на что будут похожи изменения, какую форму они примут или насколько долго они будут делаться, прежде чем попадут в новое издание книги. Просто после выхода новых выпусков ASP.NET Core MVC проверяйте хранилище GitHub для книги.

Установка .NET Core

Комплект разработки программного обеспечения .NET Core Software Development Kit (SDK) включает исполняющую среду и инструменты разработки, необходимые для построения и запуска проектов .NET. Чтобы установить .NET Core SDK в среде Windows, загрузите программу установки по ссылке `https://www.microsoft.com/net/download/thank-you/dotnet-sdk-2.1.4-windows-x64-installer`. По указанному URL находится 64-разрядная версия .NET Core SDK 2.1.4, которая используется в книге и которую вы должны установить для получения ожидаемых результатов при выполнении примеров. (Кроме того, компания Microsoft публикует программу установки только исполняющей среды, но она не устанавливает инструменты, требующиеся для проработки примеров в книге.)

Запустите программу установки и после завершения процесса установки откройте окно PowerShell или командной строки, а в нем введите команду из листинга 2.1, чтобы проверить работоспособность .NET Core.

Листинг 2.1. Тестирование .NET Core

```
dotnet --version
```

В результате выводится версия установленной исполняющей среды .NET Core. Если вы установили указанную ранее версию, тогда отобразится `2.1.4`.

Установка Visual Studio 2017

Visual Studio — традиционная среда разработки для проектов ASP.NET Core и Entity Framework Core. Загрузите программу установки по ссылке `https://www.visualstudio.com/vs`. Доступны разные редакции Visual Studio 2017, но для проработки примеров в книге вполне достаточно бесплатной редакции Community. Запустите программу установки и удостоверьтесь, что выбрана рабочая нагрузка .NET Core Cross-Platform Development (Межплатформенная разработка .NET Core), как демонстрируется на рис. 2.1.

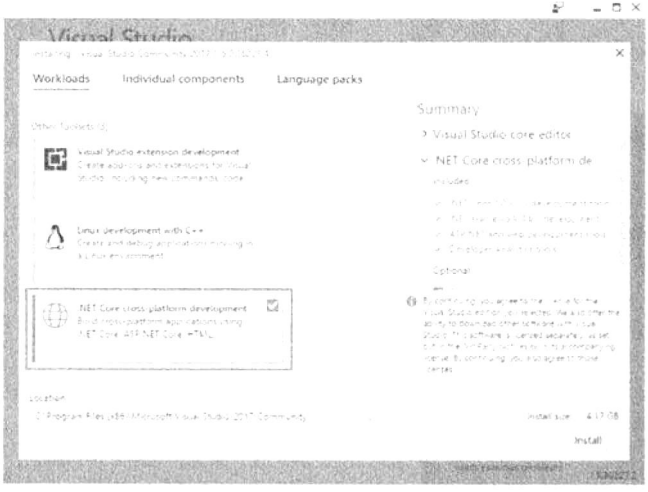

Рис. 2.1. Выбор пакетов Visual Studio

Указанная рабочая нагрузка включает версию LocalDB продукта SQL Server, которая применяется повсюду в книге, а также функциональные средства Visual Studio, требуемые для разработки с использованием ASP.NET Core MVC и Entity Framework Core. Щелкните на кнопке Install (Установить), чтобы начать процесс загрузки и установки Visual Studio.

Добавление расширений Visual Studio

При работе с проектами ASP.NET Core MVC важны два расширения Visual Studio. Первое расширение называется Razor Language Services (Службы языка Razor) и обеспечивает поддержку IntelliSense для вспомогательных функций дескрипторов, когда редактируются представления Razor. Второе расширение называется Project File Tools (Инструменты файла проекта) и предоставляет поддержку автоматического завершения при редактировании файлов `.csproj`, которая упрощает процесс добавления пакетов NuGet в проекты. (Вы можете обнаружить, что упомянутые расширения уже установлены, т.к. временами в Microsoft изменяют компоненты, добавляемые по умолчанию.)

Выберите в меню Tools (Сервис) среды Visual Studio пункт Extensions and Updates (Расширения и обновления), щелкните на категории Online (Онлайновые) и с помощью поля поиска найдите расширения. Щелкните на кнопке Download (Загрузить), чтобы загрузить файлы расширений (рис. 2.2).

Щелкните на кнопке Close (Закрыть), чтобы закрыть список расширений, и затем закройте Visual Studio, что инициирует процесс установки загруженных расширений. Вам будет предложено принять вносимые изменения и согласиться с условиями лицензии (рис. 2.3). Щелкните на кнопке Modify (Изменить) для установки расширений. После завершения процесса можно запустить Visual Studio и приступить к разработке.

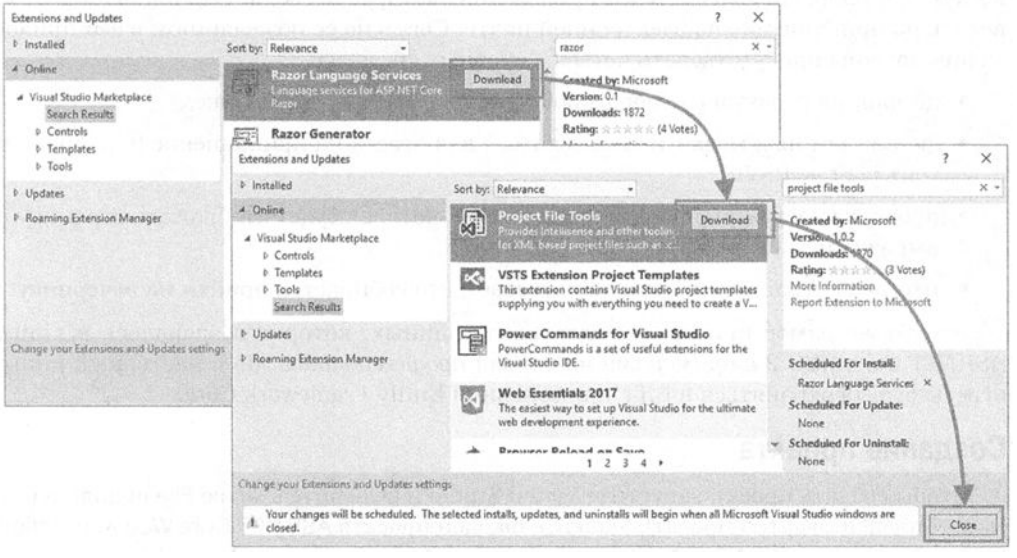

Рис. 2.2. Загрузка расширений Visual Studio

Рис. 2.3. Установка расширений Visual Studio

Создание проекта

Для ознакомления с инфраструктурой Entity Framework Core будет показано, как создать простое приложение для ввода данных, которое хранит свои данные в БД. Мы быстро пройдем через процесс создания, не вдаваясь в особые детали, чтобы вы могли ощутить, как ASP.NET Core MVC и Entity Framework Core способны работать вместе. Но не переживайте: все, что делается в этой главе, подробно объясняется в последующих главах.

Предварительная настройка

Представьте себе, что ваша подруга решила организовать вечеринку в канун нового года и попросила создать веб-приложение, которое позволяет приглашенным ответить на приглашение по электронной почте. Согласно ее пожеланиям, в веб-приложении должны присутствовать четыре основных средства:

- домашняя страница, отображающая информацию о вечеринке;
- форма, которая может использоваться для ответа на приглашение (répondez s'il vous plaît — RSVP);
- проверка достоверности для формы RSVP, которая будет отображать страницу с выражением благодарности за внимание;
- итоговая страница, которая показывает, кто собирается прийти на вечеринку.

Это то же самое приложение для ввода данных, которое создавалось в книге *ASP.NET Core MVC 2 с примерами на C# для профессионалов*, но в настоящей книге ответы будут сохраняться в БД с применением Entity Framework Core.

Создание проекта

Чтобы создать проект, запустите Visual Studio и выберите в меню File (Файл) пункт New⇨Project (Создать⇨Проект). Укажите шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), введите `PartyInvites` в поле Name (Имя) и щелкните на кнопке Browse (Обзор) для выбора подходящего местоположения, где будет храниться проект (рис. 2.4).

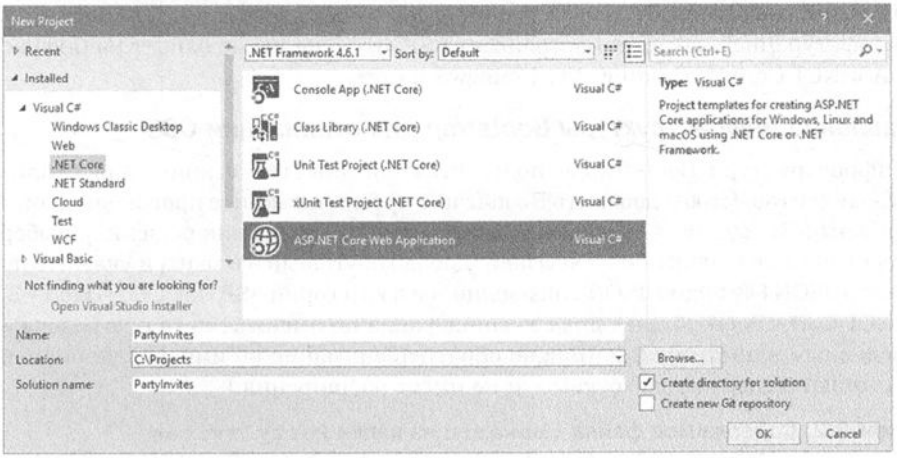

Рис. 2.4. Создание примера проекта

Совет. Проект доступен для загрузки в хранилище GitHub по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Щелкните на кнопке OK, чтобы продолжить настройку проекта. 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. 2.5. Среда Visual Studio включает шаблоны, которые настраивают инфраструктуры ASP.NET Core MVC и Entity Framework Core в проекте, но результат скрывает ряд полезных деталей.

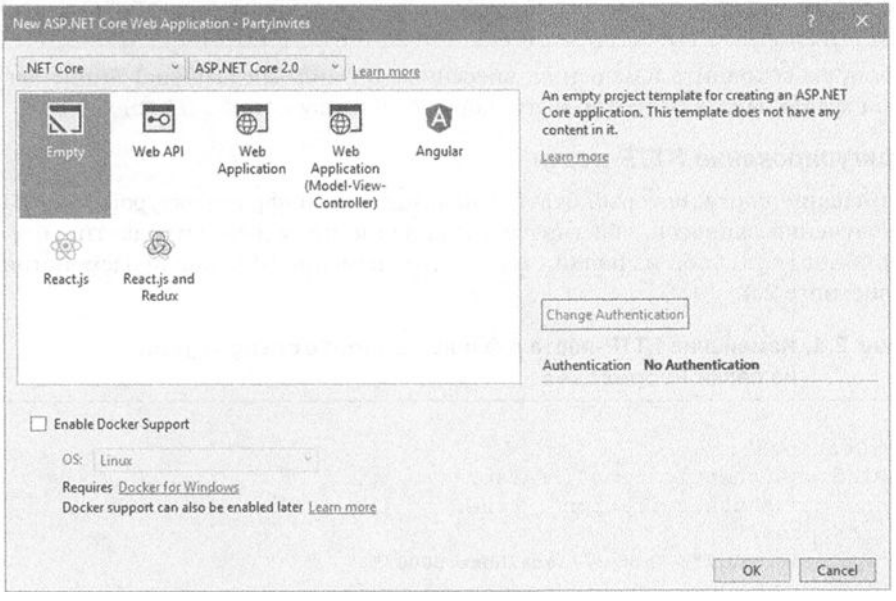

Рис. 2.5. Конфигурирование проекта ASP.NET Core

Щелкните на кнопке OK; среда Visual Studio создаст проект `PartyInvites` с базовой конфигурацией, которая настраивает ASP.NET Core, но не влияет на инфраструктуру ASP.NET Core MVC или Entity Framework Core.

Добавление инфраструктуры Bootstrap для стилизации CSS

Инфраструктура Bootstrap используется повсеместно в книге для стилизации HTML-элементов. Чтобы добавить Bootstrap в проект, щелкните правой кнопкой мыши на элементе `PartyInvites` в окне Solution Explorer (Проводник решения), выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон элемента JSON File (Файл JSON), находящийся в категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), для создания файла по имени `.bowerrc` с содержимым, приведенным в листинге 2.2. (Важно обратить внимание на имя файла: оно начинается с точки, содержит две буквы `r` и не имеет расширения.)

Листинг 2.2. Содержимое файла `.bowerrc` из папки `PartyInvites`

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

Еще раз примените шаблон JSON File для создания файла по имени `bower.json` и поместите в него содержимое, показанное в листинге 2.3.

Листинг 2.3. Содержимое файла `bower.json` из папки `PartyInvites`

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

Когда вы сохраните изменения, внесенные в файл, среда Visual Studio загрузит новую версию пакета Bootstrap и установит ее в папку `wwwroot/lib`.

Конфигурирование HTTP-порта

Изменение порта, который будет использоваться инфраструктурой ASP.NET Core для получения запросов, облегчит отслеживание примеров. Отредактируйте файл `launchSettings.json` из папки `Properties`, изменив URL, как продемонстрировано в листинге 2.4.

Листинг 2.4. Изменение HTTP-порта 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"
      }
    },
    "PartyInvites": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Указанные в файле URL применяются для конфигурирования приложения при его запуске с использованием IIS Express и в командной строке. Все примеры, рассматриваемые в книге, запускаются в командной строке, поэтому журнальные сообщения легко заметны.

Создание модели данных и классов контекста

Когда создается приложение, в котором применяются инфраструктуры ASP.NET Core MVC и Entity Framework Core, модель данных становится особенно важной. Чтобы создать класс модели для примера приложения, добавьте в проект папку `Models`, создайте в ней файл по имени `GuestResponse.cs` и поместите в него код из листинга 2.5.

Листинг 2.5. Содержимое файла `GuestResponse.cs` из папки `Models`

```
namespace PartyInvites.Models {

  public class GuestResponse {

    public long Id { get; set; }

    public string Name { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public bool? WillAttend { get; set; }
  }
}
```

Инфраструктура Entity Framework Core способна сохранять экземпляры обычных классов C# при условии, что они имеют свойство, значение которого уникально идентифицирует каждый объект; оно известно как *свойство первичного ключа*. В случае класса `GuestResponse` свойством первичного ключа является `Id`.

Средства Entity Framework Core предоставляются классом контекста БД, который идентифицирует классы моделей данных для инфраструктуры Entity Framework Core и используется для доступа к данным в БД. Чтобы создать класс контекста, добавим в папку `Models` файл по имени `DataContext.cs` с содержимым, приведенным в листинге 2.6.

Листинг 2.6. Содержимое файла `DataContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace PartyInvites.Models {

  public class DataContext : DbContext {

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

    public DbSet<GuestResponse> Responses { get; set; }
  }
}
```

При создании класса контекста БД важно включать конструктор, который принимает объект конфигурации и передает его конструктору базового класса. Для каждого класса модели данных, к которому нужен доступ, в классе контекста определено свойство, возвращающее объект `DbSet<T>` — именно через него сохраняются и извлекаются данные. Поскольку определено свойство, которое возвращает объект `DbSet<GuestResponse>`, можно сохранять и извлекать объекты `GuestResponse`.

Создание контроллера и представлений

Чтобы снабдить приложение контроллером, создайте папку `Controllers` и добавьте в нее файл по имени `HomeController.cs` с кодом из листинга 2.7.

Листинг 2.7. Содержимое файла `HomeController.cs` из папки `Controllers`

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

namespace PartyInvites.Controllers {

  public class HomeController : Controller {
    private DataContext context;

    public HomeController(DataContext ctx) => context = ctx;

    public IActionResult Index() => View();

    public IActionResult Respond() => View();

    [HttpPost]
    public IActionResult Respond(GuestResponse response) {
      context.Responses.Add(response);
      context.SaveChanges();
      return RedirectToAction(nameof(Thanks),
        new { Name = response.Name, WillAttend = response.WillAttend });
    }

    public IActionResult Thanks(GuestResponse response) => View(response);

    public IActionResult ListResponses() =>
      View(context.Responses.OrderByDescending(r => r.WillAttend));
  }
}
```

Контроллер принимает объект `DataContext` в качестве параметра конструктора и применяет его для доступа к данным, управляемым инфраструктурой Entity Framework Core. Объект `DbSet<GuestResponse>`, возвращаемый свойством `Responses` класса контекста БД, реализует интерфейс `IEnumerable<GuestResponse>`, что приводит к автоматическому запрашиванию у БД сохраненных объектов `GuestResponse` при его перечислении.

Объект `DbSet<GuestResponse>` также используется для сохранения объектов. Метод `Add()` применяется для передачи инфраструктуре Entity Framework Core объекта, подлежащего сохранению, а метод `SaveChanges()` выполняет обновление.

Совет. В более сложных проектах для доступа к функциональным средствам Entity Framework Core рекомендуется использовать паттерн REST, как объясняется в главе 10. В простом проекте этой главы контроллер работает напрямую с классом контекста БД.

Создайте папку `Views/Home` и добавьте в нее файл по имени `_Layout.cshtml`, содержимое которого показано в листинге 2.8. Файл `_Layout.cshtml` обеспечивает общую компоновку для всех других представлений в главе и содержит элемент `link`, включающий CSS-файл Bootstrap CSS.

Листинг 2.8. Содержимое файла `_Layout.cshtml` из папки `Views/Home`

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Party Invites</title>
  <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
  @RenderBody()
</body>
</html>
```

Чтобы компоновка по умолчанию применялась в представлении, создайте в папке `Views` файл по имени `_ViewStart.cshtml` с содержимым из листинга 2.9.

Листинг 2.9. Содержимое файла `_ViewStart.cshtml` из папки `Views`

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

Для создания представления, которое обеспечит посадочную страницу, добавьте в папку `Views/Home` файл по имени `Index.cshtml` с содержимым, приведенным в листинге 2.10.

Листинг 2.10. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
<div class="text-center m-4">
  <h3>We're going to have an exciting party!</h3>
  <h4>And you are invited</h4>
  <a class="btn btn-primary" asp-action="Respond">RSVP Now</a>
</div>
```

Для получения деталей запроса от пользователя добавьте в папку `Views/Home` файл по имени `Respond.cshtml` с содержимым из листинга 2.11.

Листинг 2.11. Содержимое файла `Respond.cshtml` из папки `Views/Home`

```
@model GuestResponse
<div class="bg-primary p-2 text-white text-center">
  <h2>RSVP</h2>
</div>
<form asp-action="Respond" method="post" class="m-4">
  <div class="form-group">
    <label>Your Name</label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label>Your Email</label>
    <input asp-for="Email" class="form-control" />
  </div>
  <div class="form-group">
    <label>Your Phone Number</label>
    <input asp-for="Phone" class="form-control" />
  </div>
  <div class="form-group">
    <label>Will You Attend?</label>
    <select asp-for="WillAttend" class="form-control">
      <option value="">Choose an option</option>
      <option value="true">Yes, I'll be there</option>
      <option value="false">No, I can't come</option>
    </select>
  </div>
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Submit RSVP</button>
  </div>
</form>
```

Для подтверждения ответов, предоставляемых пользователями, добавьте в папку `Views/Home` файл по имени `Thanks.cshtml`, содержимое которого показано в листинге 2.12.

Листинг 2.12. Содержимое файла `Thanks.cshtml` из папки `Views/Home`

```
@model GuestResponse
<div class="text-center mt-3">
  <h1>Thank you, @Model.Name!</h1>
  @if (Model.WillAttend == true) {
    <div>
      It's great that you're coming. The drinks are already in the fridge!
    </div>
  } else {
    <div>
      Sorry to hear that you can't make it, but thanks for letting us know.
    </div>
  }
  Click <a asp-action="ListResponses">here</a> to see who is coming.
</div>
```

Для создания финального представления добавьте в папку `Views/Home` файл по имени `ListResponses.cshtml` с содержимым из листинга 2.13, где отображается список всех ответов, полученных приложением.

Листинг 2.13. Содержимое файла `ListResponses.cshtml` из папки `Views/Home`

```
@model IEnumerable<GuestResponse>

<h3 class="bg-primary p-2 text-white text-center">Here is the list of
people who have responded</h3>

<div class="container-fluid">
  <div class="row p-1">
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Email</div>
    <div class="col font-weight-bold">Phone</div>
    <div class="col font-weight-bold">Attending</div>
  </div>
  @foreach (GuestResponse r in Model) {
    <div class="row p-1">
      <div class="col">@r.Name</div>
      <div class="col">@r.Email</div>
      <div class="col">@r.Phone</div>
      <div class="col">@(r.WillAttend == true ? "Yes" : "No")</div>
    </div>
  }
</div>
```

Чтобы завершить конфигурацию для представлений приложения, добавьте в папку `Views` файл по имени `_ViewImports.cshtml`, содержимое которого приведено в листинге 2.14.

Листинг 2.14. Содержимое файла `_ViewImports.cshtml` из папки `Views`

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

Операторы в листинге 2.14 позволяют использовать в представлениях классы из пространства имен `Models` без уточнения, а также активизируют средство вспомогательных функций дескрипторов, с помощью которого конфигурируется ряд HTML-элементов в представлениях.

Конфигурирование Entity Framework Core

Когда среда Visual Studio создает новый проект ASP.NET Core, она добавляет почти все пакеты NuGet, требующиеся для разработки посредством ASP.NET Core MVC и Entity Framework Core. Потребуется еще одно добавление, чтобы включить поддержку инструментов командной строки, которые Entity Framework Core применяет для сохранения данных. Щелкните правой кнопкой мыши на элементе `PartyInvites` в окне Solution Explorer, выберите в контекстном меню пункт Edit PartyInvites.csproj (Редактировать `PartyInvites.csproj`) и добавьте элемент конфигурации, представленный в листинге 2.15.

Листинг 2.15. Добавление пакета в файле `PartyInvites.csproj` из папки `PartyInvites`

```
<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="2.0.1" />
  </ItemGroup>

</Project>
```

Пакеты, предоставляющие инструменты командной строки, должны добавляться с использованием элемента `DotNetCliToolReference`. Среда Visual Studio предлагает инструменты для управления пакетами NuGet, но не в состоянии добавить пакет этого вида. После сохранения файла Visual Studio загрузит пакет и добавит его к проекту.

Конфигурирование строки подключения

При работе с базой данных вы обязаны предоставить строку подключения, которая сообщает инфраструктуре Entity Framework Core, каким образом подключаться к базе данных, и часто содержит дополнительную информацию, такую как учетные данные аутентификации. Щелкните правой кнопкой мыши на элементе `PartyInvites` в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон элемента ASP.NET Configuration File (Файл конфигурации ASP.NET); удостоверьтесь, что поле Name (Имя) установлено в `appsettings.json` (рис. 2.6).

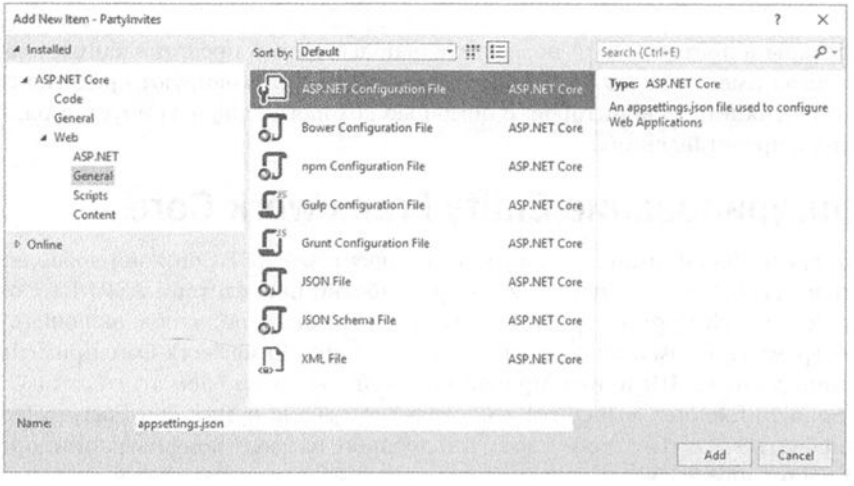

Рис. 2.6. Создание файла конфигурации приложения

Щелкните на кнопке Add (Добавить), чтобы создать файл, и измените строку подключения, как показано в листинге 2.16.

Листинг 2.16. Определение строки подключения в файле `appsettings.json` из папки `PartyInvites`

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=PartyInvites"
  }
}
```

Эта строка подключения сообщает инфраструктуре Entity Framework Core о применении базы данных по имени `PartyInvites`, используя средство LocalDB, которое было установлено вместе с Visual Studio.

Конфигурирование класса `Startup`

Следующий шаг предусматривает конфигурирование приложения для настройки ASP.NET Core MVC и Entity Framework Core путем добавления в класс `Startup` операторов, приведенных в листинге 2.17.

Листинг 2.17. Конфигурирование приложения в файле `Startup.cs` из папки `PartyInvites`

```
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.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using PartyInvites.Models;

namespace PartyInvites {

  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<DataContext>(options =>
        options.UseSqlServer(conString));
    }

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

```
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

Изменения загружают настройки в файле `appsettings.json` и применяют их для конфигурирования класса контекста БД. В результате он будет использоваться инфраструктурой Entity Framework Core и станет доступным как служба через средство внедрения зависимостей, которое позволяет другим частям приложения легко получать объект контекста и обращаться к функциональным средствам Entity Framework Core.

Подготовка базы данных

Инфраструктура Entity Framework Core должна создать и сконфигурировать базу данных, чтобы ее можно было применять для хранения объектов `GuestResponse`. Это делается за счет создания *миграции*. Инфраструктура Entity Framework Core инспектирует модель данных приложения и выясняет, каким образом она может быть сохранена в реляционной базе данных. Результат — миграция — содержит набор инструкций, которые класс поставщика транслирует в команды SQL, указывающие серверу баз данных на необходимость создания БД для Entity Framework Core. Работа миграций подробно объясняется в главе 12.

Миграции создаются и применяются с использованием инструментов командной строки. Откройте окно PowerShell или командной строки, перейдите в папку проекта `PartyInvites` (папка, содержащая файлы `bower.json` и `Startup.cs`) и введите команду из листинга 2.18.

Листинг 2.18. Создание новой миграции

```
dotnet ef migrations add Initial
```

Чтобы применить миграцию и создать БД, выполните в папке `PartyInvites` команду из листинга 2.19.

Листинг 2.19. Применение миграции

```
dotnet ef database update
```

Тестирование приложения

Для запуска приложения введите в папке `PartyInvites` команду, показанную в листинге 2.20. Несмотря на то что приложения можно запускать с использованием Visual Studio, многие примеры в книге полагаются на журнальные сообщения, генерируемые приложениями, а их легче просматривать, когда приложение запускается в командной строке; такой подход принят во всех последующих главах.

Листинг 2.20. Запуск приложения

```
dotnet run
```

После того как приложение запустилось, откройте новое окно браузера и перейдите по ссылке `http://localhost:5000`; результат приведен на рис. 2.7.

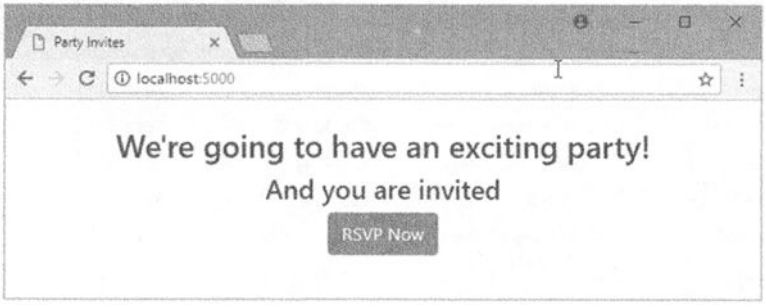

Рис. 2.7. Выполнение примера приложения

Щелкните на кнопке RSVP Now (Ответить на приглашение), заполните форму и щелкните на кнопке Submit RSVP (Отправить RSVP). Вы увидите ответ, показанный на рис. 2.8.

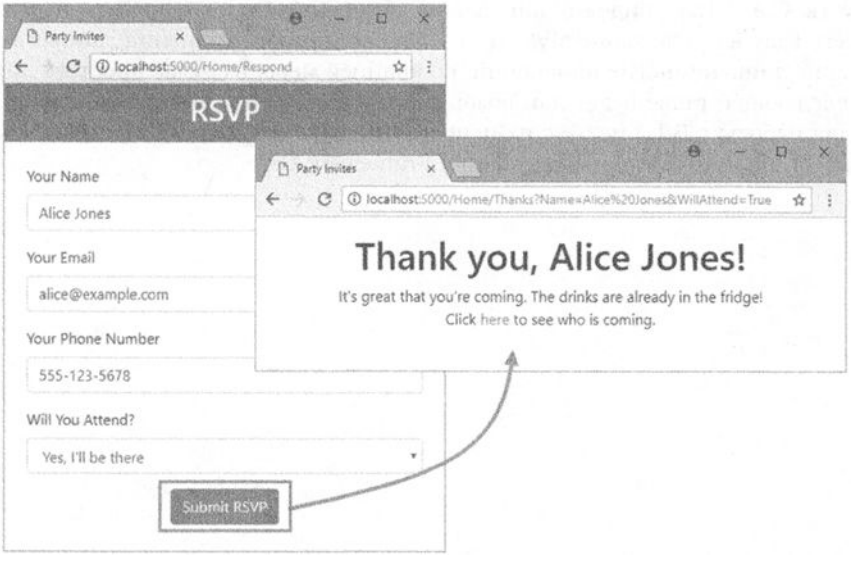

Рис. 2.8. Создание ответа

Щелкните на ссылке, чтобы просмотреть список ответов. Отобразится содержимое, подобное представленному на рис. 2.9, где присутствуют дополнительные созданные ответы.

Объекты `GuestResponse`, которые создаются связывателем моделей MVC из значений в HTTP-запросах `POST`, сохраняются в БД. Когда вы требуете список ответов, у БД запрашиваются данные и инфраструктура Entity Framework Core применяет результаты для создания последовательности объектов `GuestResponse`, которые используются для генерации списка.

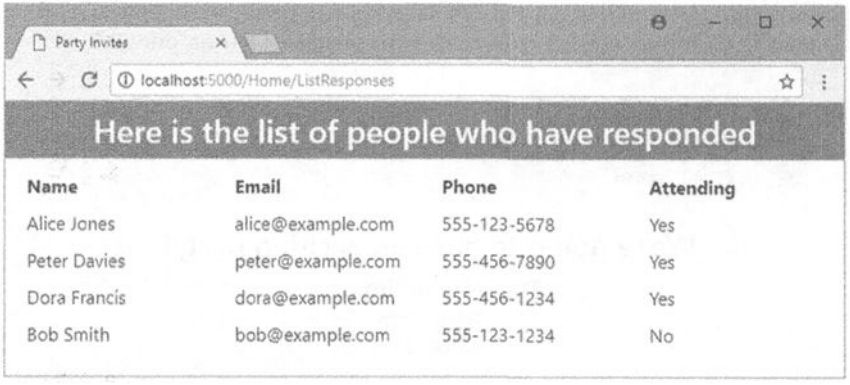

Рис. 2.9. Отображение списка ответов

Резюме

В главе был создан новый проект ASP.NET Core для простого приложения ввода данных, которое сохраняет свои данные в БД типа LocalDB с применением Entity Framework Core. Вы увидели, насколько легко добавить инфраструктуру Entity Framework Core в приложение MVC, и что для постоянного хранения данных требуется внесение лишь немногих изменений, по крайней мере, в случае простого приложения. В следующей главе будет дан обзор инструментов, предлагаемых средой Visual Studio для работы с БД, вместе с их использованием при объяснении самых важных SQL-команд, на которые опирается Entity Framework Core.

ГЛАВА 3

Работа с базами данных

Использование инфраструктуры Entity Framework Core облегчает взаимодействие с базами данных, но вовсе не освобождает разработчика от необходимости понимания, как они функционируют и каким образом действия, выполняемые приложением, транслируются в SQL-команды, особенно когда получаются непредвиденные результаты. В этой главе будут рассмотрены инструменты, которые среда Visual Studio предлагает для исследования баз данных, и показано, как выполнять SQL-команды различных видов. Для применения Entity Framework Core вы не обязаны быть экспертом в языке SQL, но понимание его основ может быть полезно в случае, если получить запланированный исход не удалось.

Выбор сервера баз данных и пакета поставщика

При выборе сервера баз данных ошибиться трудно, поскольку все доступные варианты являются высококачественными программными продуктами. Не играет роли, предпочитаете вы коммерческие продукты или продукты с открытым кодом, равно как хотите вы запускать собственный сервер либо пользоваться облаком.

Если у вас уже есть сервер баз данных, возможно из-за соблюдения корпоративного стандарта на закупки или наличия лицензионного договора на площадке, тогда с ним и нужно работать. Инфраструктура Entity Framework Core сглаживает отличия между серверами баз данных, поэтому действительно не имеет значения, какой сервер вы будете использовать. Безусловно, не стоит бороться с технологическими стандартами, принятыми в компании, если только вы не имеете очень специфические требования.

Если сервера баз данных пока нет, то доступно предостаточно хороших вариантов. Чаще всего применяется Microsoft SQL Server из-за широкого диапазона планов ценообразования, включая бесплатные опции для разработчиков и небольших проектов и версии с размещением в облаке Azure. В примерах, приводимых в книге, используется версия SQL Server для разработчиков, не требующая конфигурирования, которая известна как LocalDB.

(Я не получаю какого-либо вознаграждения за рекомендацию SQL Server и любых других продуктов, применяемых или упоминаемых в книге. Использование указываемого программного обеспечения важно для следования примерам, но в реальных проектах вы совершенно не ограничены в своем выборе.)

Если вы предпочитаете продукты с открытым кодом, тогда великолепным вариантом будет продукт MySQL, хотя он и поддерживается компанией Oracle, имеющей неоднозначный послужной список в сфере открытого кода. MariaDB является ответвлением проекта MySQL, которое не касается Oracle, но направлено на обеспечение совместимости. Есть много поставщиков, предлагающих MySQL или производные продукты в виде размещаемых/облачных служб, скажем, Amazon Web Services и Microsoft Azure.

Определившись с сервером баз данных, вы можете выбрать поставщика баз данных для применения с инфраструктурой Entity Framework Core. По ссылке `https://docs.microsoft.com/ru-ru/ef/core/providers/` предлагается список поставщиков для самых популярных баз данных. Большинство пакетов поставщиков бесплатны, но доступны и коммерческие продукты. Если вы хотите использовать сервер баз данных Oracle (коммерческий продукт, не MySQL), тогда вам понадобится лицензия для стороннего поставщика, т.к. компания Oracle еще не выпустила собственный пакет.

Подготовительные шаги

В настоящей главе мы продолжим работу с проектом `PartyInvites`, созданным в главе 2. Чтобы выполнить подготовительные шаги, откройте окно PowerShell или командной строки, перейдите в папку проекта `PartyInvites` (папку, содержащую файл `bower.json`) и введите команды из листинга 3.1, которые удалят и воссоздадут применяемую БД, гарантируя тем самым получение ожидаемых результатов при проработке примеров главы.

Листинг 3.1. Переустановка БД

```
dotnet ef database drop --force
dotnet ef database update
```

Добавьте в файл `appsettings.json` операторы конфигурации, показанные в листинге 3.2. Они отключают журнальные сообщения для всех пакетов .NET кроме Entity Framework Core, что облегчит отслеживание примеров.

Листинг 3.2. Конфигурирование журнальных сообщений в файле `appsettings.json` из папки `PartyInvites`

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

Такая конфигурация ведения журнала позволит видеть сообщения, генерируемые Entity Framework Core, которые покажут SQL-команды, отправляемые БД, и предотвратит их потерю в потоке других сообщений.

Запустите приложение, выполнив команду `dotnet run` в папке проекта, и перейдите по ссылке `http://localhost:5000` в окне браузера. Щелкните на кнопке RSVP Now (Ответить на приглашение) и создайте четыре ответа RSVP, используя значения из табл. 3.1.

Таблица 3.1. Значения данных, требующиеся в приложении

Имя	Адрес электронной почты	Телефонный номер	Примет ли участие
Alice Jones	alice@example.com	555-123-5678	Yes (Да)
Peter Davies	peter@example.com	555-456-7890	Yes (Да)
Dora Francis	dora@example.com	555-456-1234	Yes (Да)
Bob Smith	bob@example.com	555-123-1234	No (Нет)

После добавления ответов перейдите по ссылке `http://localhost:5000/home/listresponses`; вы увидите список, показанный на рис. 3.1.

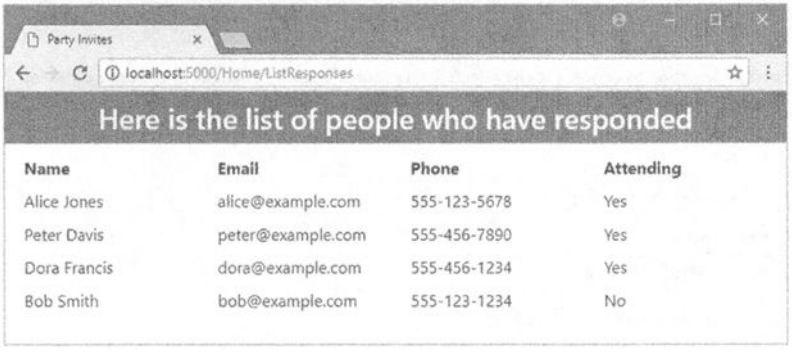

Рис. 3.1. Добавление данных в примере приложения

Исследование базы данных

Среда Visual Studio включает набор инструментов, позволяющих исследовать базы данных, с которыми приходится работать. Выберите в меню Tools (Сервис) среды Visual Studio пункт Connect to Database (Подключиться к базе данных). Откроется диалоговое окно, представленное на рис. 3.2, которое даст возможность выбрать тип сервера баз данных для подключения.

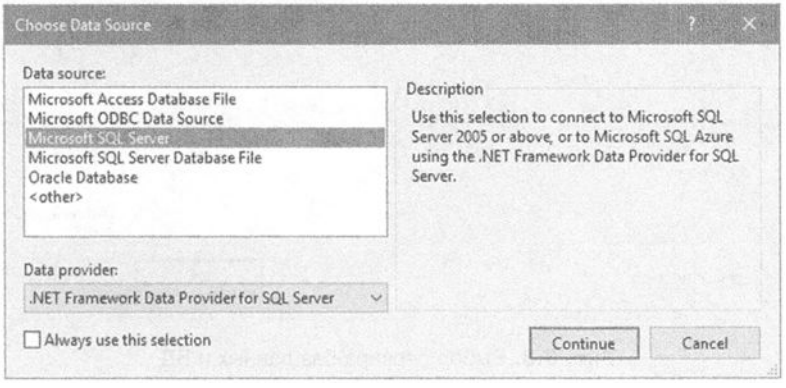

Рис. 3.2. Выбор типа сервера баз данных

Во всех примерах, рассматриваемых в книге, применяется LocalDB — версия Microsoft SQL Server, которая была установлена как часть рабочей нагрузки Visual Studio в главе 2 и предназначена для использования разработчиками. Она реализует все ключевые возможности SQL Server, но не требует конфигурирования и лицензирования для производственных систем. Выберите в списке вариант Microsoft SQL Server и щелкните на кнопке Continue (Продолжить).

Следующее диалоговое окно позволяет указать детали сервера баз данных и БД, с которой нужно работать (рис. 3.3). Введите `(localdb)\MSSQLLocalDB` в поле Server Name (Имя сервера) и возле выбранного переключателя Select or Enter a Database Name (Выберите или введите имя базы данных) выберите `PartyInvites`, как показано на рис. 3.3.

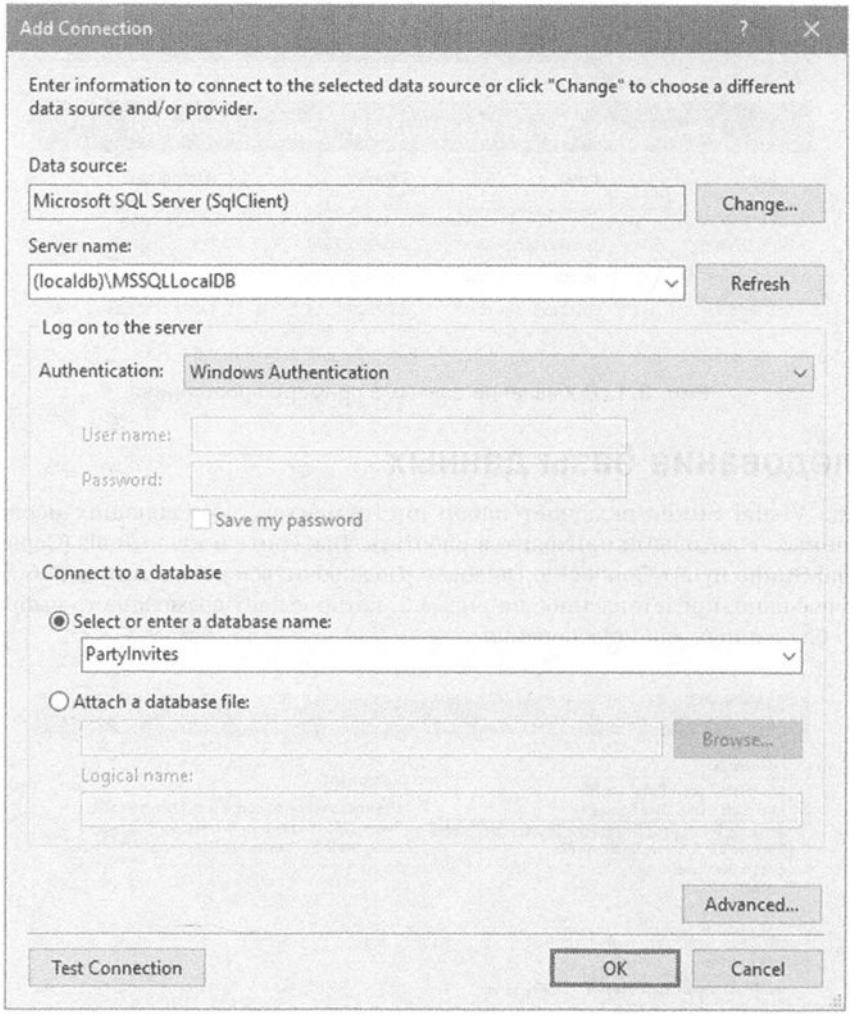

Рис. 3.3. Выбор сервера баз данных и БД

Совет. Строка, требующаяся для подключения к LocalDB, может вызвать путаницу. Первой частью является `localdb` в круглых скобках. Затем идет одиночный символ `\`, за которым следует `MS_SQL_LocalDB`, но без символов подчеркивания: `(localdb)\MSSQLLocalDB`. Дополнительный символ `\` требуется при указании строк подключения в файле `appsetting.json`, потому что обратная косая черта имеет специальное значение, которое должно быть отменено: `(localdb)\\MSSQLLocalDB`.

Щелкните на кнопке OK; среда Visual Studio подключится к серверу баз данных и отобразит детали в окне Server Explorer (Проводник сервера), которое также можно открыть, выбрав пункт меню View⇨Server Explorer (Вид⇨Проводник сервера). В разделе Data Connections (Подключения к данным) вы увидите элемент для `PartyInvites`, который можно развернуть и просмотреть подробные сведения о БД (рис. 3.4).

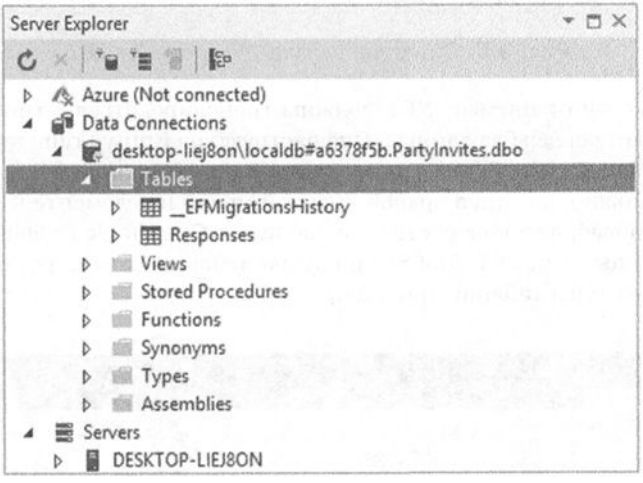

Рис. 3.4. Элемент БД в окне Server Explorer

Исследование таблиц базы данных

Развернув элемент Tables (Таблицы), вы увидите два элемента. Таблица `__EFMigrationsHistory` используется для отслеживания миграций, которые были применены к базе данных с целью ее поддержания в синхронизированном состоянии с моделью данных приложения. Работа миграций подробно обсуждается в главе 13; в настоящей же главе эта таблица интереса не представляет. Таблица `Responses` используется для хранения объектов `GuestResponse` приложения. Разверните таблицу `Responses` в окне Server Explorer, и вы увидите ее столбцы (рис. 3.5).

Вы можете заметить, как Entity Framework Core использует детали из приложения для создания таблицы, которая будет хранить объекты `GuestResponse`. Имя таблицы берется из свойства класса контекста БД, а имена столбцов соответствуют свойствам, определенным в классе `GuestResponse`. Такие решения обусловлены соглашениями, которым по умолчанию следует Entity Framework Core; в части II книги объясняется, как их переопределить.

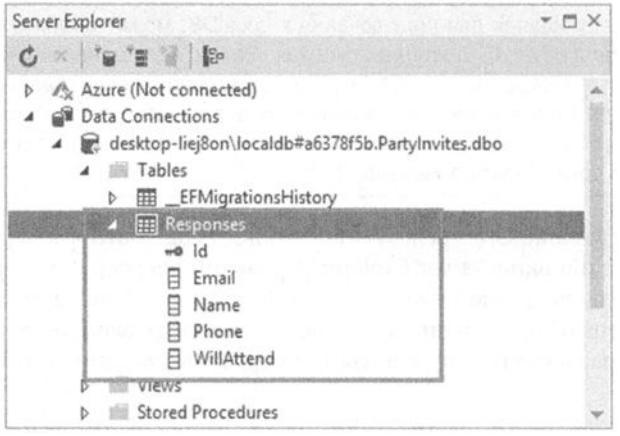

Рис. 3.5. Структура таблицы `Responses` в БД

Типы данных, применяемые .NET, должны транслироваться в типы данных, которые поддерживает сервер баз данных. Инфраструктура Entity Framework Core выясняет, какие типы баз данных будут использоваться, при создании БД, и ознакомиться с ее решениями можно, щелкнув правой кнопкой мыши на элементе `Responses` в окне Server Explorer и выбрав в контекстном меню пункт Open Table Definition (Открыть определение таблицы). Среда Visual Studio откроет новую панель редактора, содержащую детали структуры таблицы (рис. 3.6).

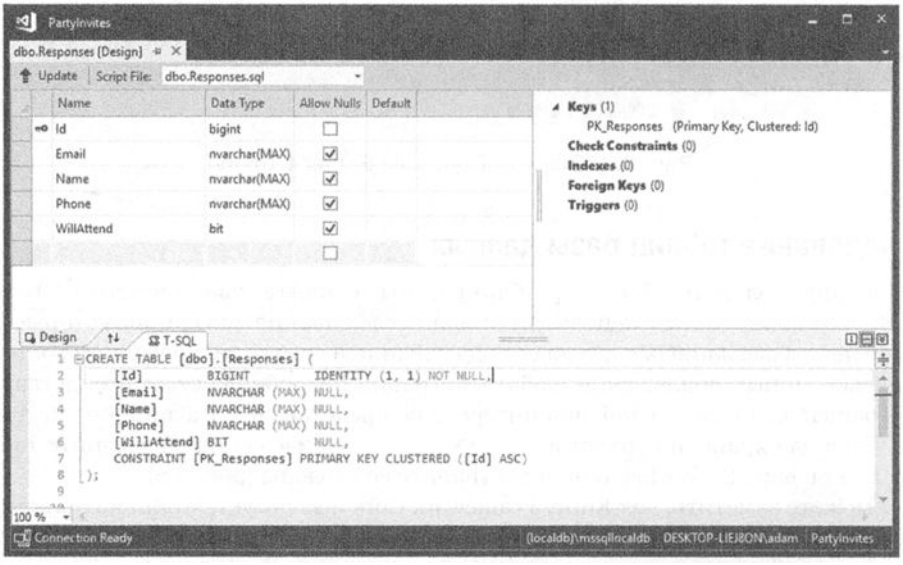

Рис. 3.6. Исследование структуры таблицы БД

Здесь видны типы данных SQL, которые были выбраны для представления значений каждого свойства `GuestResponse`. В выборе наиболее подходящих типов данных SQL инфраструктура Entity Framework Core полагается на поставщик БД, и типы, вы-

бираемые для разных серверов баз данных, могут отличаться. Для SQL Server свойство `Id` будет храниться как SQL-тип `bigint`, соответствующий .NET-типу `long`; свойство `WillAttend` — как SQL-тип `bit`, который соответствует .NET-типу `bool`, допускающему значения `null`, а остальные свойства — как SQL-тип `nvarchar(MAX)`, соответствующий .NET-типу `string`.

Совет. Для большинства проектов типы данных, которые выбирает Entity Framework Core, будут совершенно приемлемыми, но вы можете указать другие типы с применением средств, описанных в главе 21.

Исследование содержимого базы данных

Щелкните правой кнопкой мыши на имени таблицы в окне Server Explorer и выберите в контекстном меню пункт Show Table Data (Показать табличные данные), чтобы просмотреть данные в таблице. Среда Visual Studio выполнит запрос к базе данных и отобразит результаты (рис. 3.7).

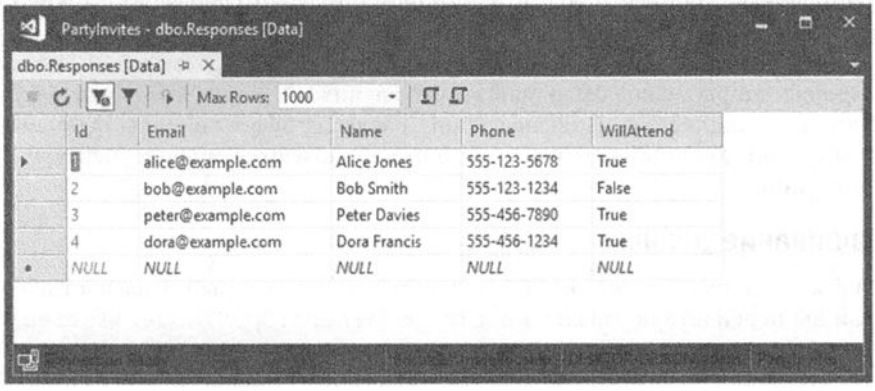

Рис. 3.7. Отображение данных в таблице БД

Сетка, используемая средой Visual Studio для отображения данных, поддерживает редактирование, т.е. вы можете обновлять существующие значения данных и добавлять в таблицу новые строки, не применяя SQL напрямую.

Добавьте новый ответ, используя значения из табл. 3.2, чтобы заполнить поля в нижней строке сетки, и затем нажмите клавишу <Tab>. У вас не будет возможности вводить значение для столбца `Id`, но сервер баз данных самостоятельно присвоит значение столбцу `Id` при добавлении данных в БД.

Таблица 3.2. Значения данных для добавления строки в таблицу БД

Email	Name	Phone	WillAttend
jane@example.com	Jane Marshall	555-123-1212	True

Когда вы нажмете клавишу <Tab>, среда Visual Studio добавит новые данные в БД. Вы можете увидеть новые данные в приложении, перейдя по ссылке `http://localhost:5000/home/listresponses` (рис. 3.8).

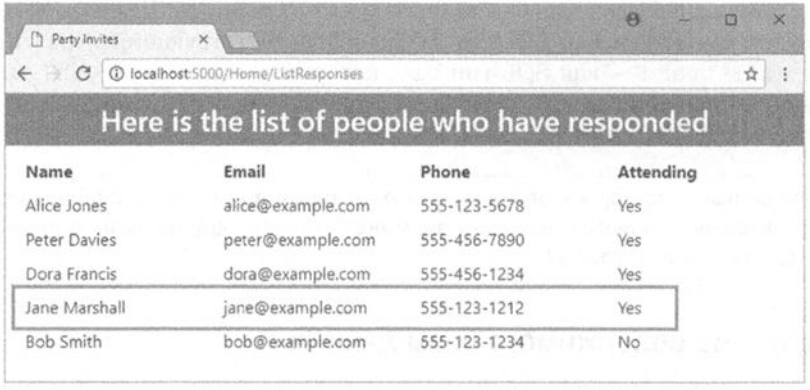

Рис. 3.8. Новые данные, отображаемые приложением

Введение в SQL

Зачастую может принести пользу понимание того, каким образом Entity Framework Core транслирует действия приложения в команды для сервера баз данных. Для применения Entity Framework Core вы не обязаны быть экспертом в SQL, но полезно знать основы, чтобы можно было понять, что случилось, если полученные результаты отличаются от ожидаемых. В последующих разделах будут описаны основные SQL-команды и приведены объяснения, как они используются инфраструктурой Entity Framework Core.

Запрашивание данных

Фундаментальным средством SQL является запрос, который извлекает данные из БД. Если вы перейдете по ссылке `http://localhost:5000/home/listresponses` и просмотрите журнальные сообщения, сгенерированные приложением, то сможете увидеть SQL-команду, которую инфраструктура Entity Framework Core применяла для получения данных из БД:

```
...
SELECT [r].[Id], [r].[Email], [r].[Name], [r].[Phone], [r].[WillAttend]
FROM [Responses] AS [r]
ORDER BY [r].[WillAttend] DESC
...
```

Если вам знаком язык LINQ, тогда простой взгляд на эту SQL-команду позволит уловить смысл того, что она делает, но ее стоит исследовать чуть глубже, чтобы понять работу, выполняемую Entity Framework Core.

Запрашивание базы данных вручную

Вы можете запрашивать БД напрямую с использованием окна Server Explorer среды Visual Studio, щелкнув правой кнопкой мыши на имени таблицы и выбрав в контекстном меню пункт New Query (Новый запрос). При работе с SQL Server предпочтительнее применять другое средство Visual Studio, для чего выбрать пункт New Query в меню Tools⇨SQL Server (Сервис⇨SQL Server). Когда откроется диалоговое окно Connect (Подключение), показанное на рис. 3.9, введите `(localdb)\MSSQLLocalDB` в поле Server Name (Имя сервера), оставьте <default> (<по умолчанию>) в поле Database Name (Имя базы данных) и щелкните на кнопке Connect (Подключиться).

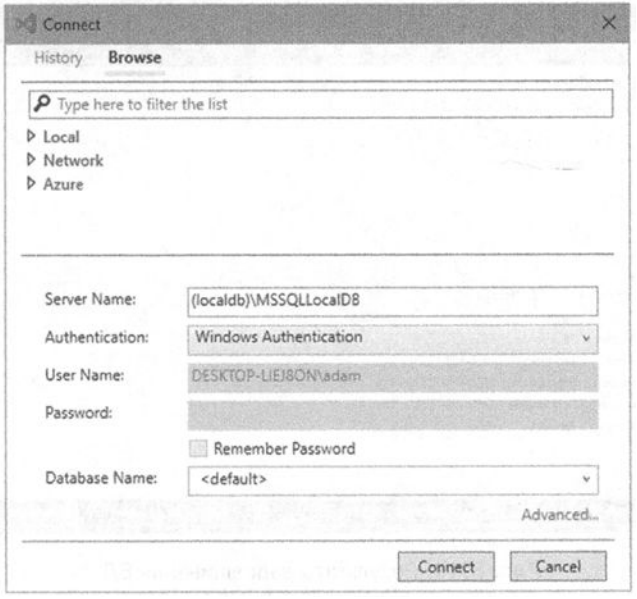

Рис. 3.9. Подключение к БД

Совет. После подключения к БД настройки подключения сохраняются в таблице `History` диалогового окна Connect и могут использоваться для подключения в будущем, не вводя детали повторно.

Щелчок на кнопке Connect приводит к тому, что Visual Studio подключится к серверу баз данных и откроет новую панель редактора, в которой можно вводить SQL-команды. Введите команды, представленные в листинге 3.3.

Листинг 3.3. Элементарный SQL-запрос

```
USE PartyInvites
SELECT * FROM Responses
```

Чтобы выполнить команды, щелкните на кнопке с изображением зеленой стрелки в левом верхнем углу панели редактора или щелкните правой кнопкой мыши в окне и выберите в контекстном меню пункт Execute (Выполнить). Среда Visual Studio отправит SQL-команду серверу баз данных и отобразит результаты (рис. 3.10).

Поскольку это первая SQL-команда, которую вы выполнили напрямую, необходимо разобрать ее на части и объяснить каждую из них. Вот одна часть:

```
...
USE PartyInvites
...
```

Показанная часть выбирает БД по имени `PartyInvites`. Сервер баз данных может управлять многими базами данных и важно выбрать БД, с которой нужно работать. (Вы можете выбрать БД, когда создаете подключение, или применить раскрывающийся список в верхней части панели редактора, но лучше производить выбор явным образом.)

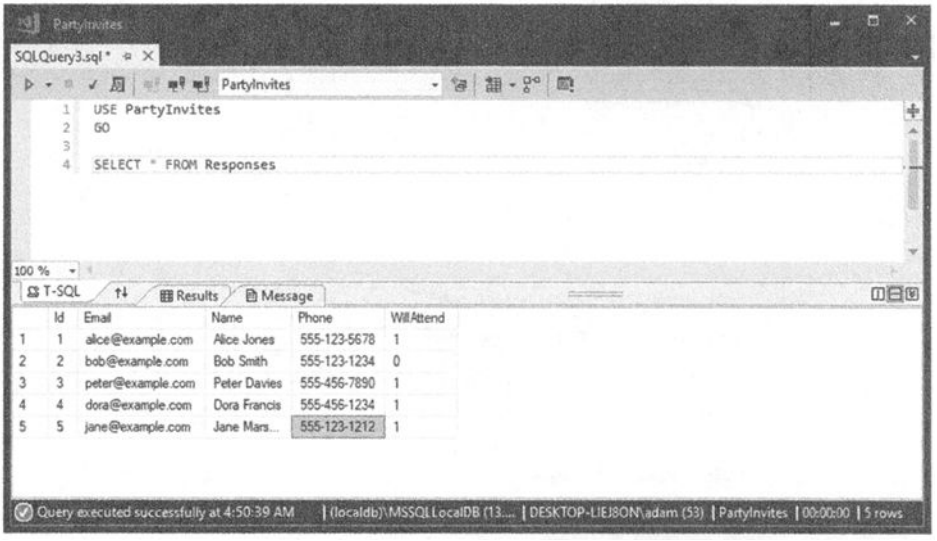

Рис. 3.10. Результаты запрашивания БД

Второй частью SQL-команды из листинга 3.3 является фактический запрос:

```
...
SELECT * FROM Responses
...
```

Ключевым словом `SELECT` обозначается SQL-запрос. Ключевое слово `FROM` указывает таблицу БД, чьи данные требуются. Символ звездочки устанавливает, что запрос должен возвращать значения для всех столбцов в таблице. Подводя итоги, полный запрос сообщает серверу баз данных о предоставлении всех столбцов из таблицы `Responses`. Запрос дает результаты, показанные в табл. 3.3.

Таблица 3.3. Результаты, получаемые после выполнения элементарного запроса

Id	Email	Name	Phone	WillAttend
1	alice@example.com	Alice Jones	555-123-5678	1
2	bob@example.com	Bob Smith	555-123-1234	0
3	peter@example.com	Peter Davies	555-456-7890	1
4	dora@example.com	Dora Francis	555-456-1234	1
5	jane@example.com	Jane Marshall	555-123-1212	1

Фильтрация данных

По умолчанию запрос будет возвращать все строки в таблице. Такой результат полностью приемлем для приложений, хранящих небольшие объемы данных, таких как пример проекта, но в большинстве реальных проектов понадобится запрашивать подмножество данных. Ключевое слово `WHERE` используется для выбора специфических строк из таблицы БД. Введите в панели редактора SQL-команду, приведенную в листинге 3.4, чтобы посмотреть, как работает `WHERE`.

Соглашения, принятые в языке SQL

Вы заметите, что в главе ключевые слова SQL записываются буквами верхнего регистра. Поступать так вовсе не обязательно, но это рекомендуемая практика, которая содействует упрощению чтения сложных SQL-команд. Мир SQL полон соглашений и предпочтений — в точности как разработка на языке C# — и вы неминуемо будете встречать администраторов баз данных и разработчиков, настаивающих на определенном стиле. Серверы баз данных умеют разбирать SQL-команды, и я советую придерживаться такого стиля, который легко воспринимается вами и коллегами, даже если написание команд имеет тенденцию соблюдать соглашения, принятые для кода C#, а не связанные с кодом SQL.

Листинг 3.4. Запрашивание избранных строк

```
USE PartyInvites

SELECT * FROM Responses
WHERE WillAttend = 1
```

За ключевым словом `WHERE` следует выражение, дающее совпадение для строк, которые должны быть включены в результаты. В данном случае указано, что строки, которые имеют значение `WillAttend`, равное `1`, должны включаться в результаты, а остальные строки — нет. Выполните команду и вы увидите только совпадающие ответы (табл. 3.4).

Таблица 3.3. Результаты, получаемые после выполнения элементарного запроса

Id	Email	Name	Phone	WillAttend
1	alice@example.com	Alice Jones	555-123-5678	1
3	peter@example.com	Peter Davies	555-456-7890	1
4	dora@example.com	Dora Francis	555-456-1234	1
5	jane@example.com	Jane Marshall	555-123-1212	1

Выбор и упорядочение столбцов

Звездочка (`*`) в запросе требует наличия всех столбцов у строк, включаемых в результаты. Все столбцы нужны не всегда, к тому же может понадобиться указать порядок следования столбцов в результате. Запрос SQL может быть выборочным в отношении столбцов, которые включаются в результаты (листинг 3.5).

Листинг 3.5. Пример выбора и упорядочения столбцов

```
USE PartyInvites

SELECT Id, Name, Email FROM Responses
WHERE WillAttend = 'true'
```

В запросе указано, что необходимы только значения для столбцов `Id`, `Name` и `Email`. Результаты выполнения запроса можно видеть в табл. 3.5, а значения остальных столбцов исключены. Обратите внимание, что результаты представлены в порядке, в котором были указаны столбцы.

Таблица 3.5. Результаты выбора столбцов

Id	Name	Email
1	Alice Jones	alice@example.com
3	Peter Davies	peter@example.com
4	Dora Francis	dora@example.com
5	Jane Marshall	jane@example.com

Упорядочение строк

С помощью ключевых слов `ORDER BY` можно указать порядок, в котором будут возвращаться совпадающие строки в результатах. В листинге 3.6 посредством `ORDER BY` серверу баз данных сообщается о том, что результаты должны быть упорядочены по значению столбца `Email`.

Листинг 3.6. Упорядочивание результатов

```
USE PartyInvites

SELECT Id, Name, Email FROM Responses
WHERE WillAttend = 'true'
ORDER BY Email
```

После выполнения запроса результаты будут упорядочены по значению столбца `Email` (табл. 3.6).

Таблица 3.6. Результаты упорядочения строк

Id	Name	Email
1	Alice Jones	alice@example.com
4	Dora Francis	dora@example.com
5	Jane Marshall	jane@example.com
3	Peter Davies	peter@example.com

Исследование запроса Entity Framework Core

Запросы SQL обладают многими характеристиками, но показанных в предшествующих разделах свойств вполне достаточно, чтобы получить представление о том, как данные запрашиваются из БД. В запросах, выпускаемых инфраструктурой Entity Framework Core, существует несколько незначительных отличий, которые заслуживают пояснения. В листинге 3.7 запрос LINQ, применяемый в методе действия `ListResponses()` контроллера `Home`, изменяется, чтобы стать избирательнее в отношении запрашиваемых данных.

Листинг 3.7. Изменение запроса LINQ в файле `HomeController.cs` из папки `Controllers`

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

```
namespace PartyInvites.Controllers {
  public class HomeController : Controller {
    private DataContext context;
    public HomeController(DataContext ctx) => context = ctx;
    public IActionResult Index() => View();
    public IActionResult Respond() => View();

    [HttpPost]
    public IActionResult Respond(GuestResponse response) {
      context.Responses.Add(response);
      context.SaveChanges();
      return RedirectToAction(nameof(Thanks),
        new { Name = response.Name, WillAttend = response.WillAttend });
    }
    public IActionResult Thanks(GuestResponse response) => View(response);
    public IActionResult ListResponses() =>
     View(context.Responses
       .Where(r => r.WillAttend == true)
       .OrderBy(r => r.Email));
  }
}
```

Запустите приложение, используя команду `dotnet run` в папке проекта, и перейдите по ссылке `http://localhost:5000/home/listresponses`. Отображаемая пользователю HTML-разметка будет содержать только ответы от тех, кто посетит вечеринку, упорядоченные по их адресам электронной почты (рис. 3.11).

Рис. 3.11. Применение более избирательного запроса в примере приложения

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то заметите SQL-команду, которую инфраструктура Entity Framework Core выработала на основе запроса LINQ:

```
...
SELECT [r].[Id], [r].[Email], [r].[Name], [r].[Phone], [r].[WillAttend]
FROM [Responses] AS [r]
WHERE [r].[WillAttend] = 1
ORDER BY [r].[Email]
...
```

Как здесь видно, структура запроса следует тому же самому шаблону, что и запросы, построенные в предшествующих разделах. Использование квадратных скобок ([и]) позволяет именам столбцов содержать специальные символы, такие как пробелы. Для имеющейся модели данных это не требуется, но Entity Framework Core поступает так в любом случае.

Ключевое слово AS применяется для создания временного псевдонима. Инфраструктура Entity Framework Core использует AS для ссылки на таблицу Responses с помощью r. В простом запросе такой прием не дает особых преимуществ, но может быть полезен в более сложных запросах, которые объединяют данные из множества таблиц.

Понятие параметров запроса

При исследовании запросов, создаваемых инфраструктурой Entity Framework Core, вы можете заметить еще одно отличие. В целях демонстрации измените запрос, применяемый методом действия ListResponses(), как показано в листинге 3.8.

Листинг 3.8. Изменение запроса LINQ Query в файле HomeController.cs из папки Controllers

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

namespace PartyInvites.Controllers {

  public class HomeController : Controller {
    private DataContext context;

    public HomeController(DataContext ctx) => context = ctx;

    public IActionResult Index() => View();

    public IActionResult Respond() => View();

    [HttpPost]
    public IActionResult Respond(GuestResponse response) {
      context.Responses.Add(response);
      context.SaveChanges();
      return RedirectToAction(nameof(Thanks),
        new { Name = response.Name, WillAttend = response.WillAttend });
    }

    public IActionResult Thanks(GuestResponse response) => View(response);

    public IActionResult ListResponses(string searchTerm = "555-123-5678") =>
      View(context.Responses
        .Where(r => r.Phone == searchTerm)
        .OrderBy(r => r.Email));
  }
}
```

Запрос LINQ использует метод Where() для выбора объектов GuestResponse, чьи значения Phone совпадают с поисковым параметром. Если вы запустите приложение, перейдете по ссылке http://localhost:5000/home/listresponses и просмотрите журнальные сообщения, сгенерированные приложением, то увидите, как такой запрос LINQ был транслирован в SQL:

```
...
SELECT [r].[Id], [r].[Email], [r].[Name], [r].[Phone], [r].[WillAttend]
FROM [Responses] AS [r]
WHERE [r].[Phone] = @__searchTerm_0
ORDER BY [r].[Email]
...
```

Символ @ применяется для обозначения параметра в SQL-запросе. Параметры используются, когда Entity Framework Core имеет дело с переменными; они позволяют серверу баз данных распознавать похожие запросы и улучшать показатели производительности, обрабатывая их аналогичным образом. Параметры также обеспечивают защиту против атак внедрением SQL-кода, когда пользователь вводит значение данных, которое может интерпретироваться как часть SQL-команды.

Применение параметров — хорошая идея, особенно из-за того, что инфраструктура Entity Framework Core не знает, можно ли доверять источнику значения данных. Вместе с тем наличие параметров затрудняет восприятие SQL-команды, отправляемой БД, поскольку по умолчанию значение параметра в журнальных сообщениях не отображается. Большую часть времени это не проблема, т.к. нас интересует структура запроса, но в листинге 3.9 конфигурация примера приложения изменена, чтобы инфраструктура Entity Framework Core включала значения параметров в генерируемые журнальные сообщения.

Внимание! Не оставляйте такую настройку включенной в производственной версии, потому что тогда уязвимые данные попадут в журнальные файлы приложения.

Листинг 3.9. Включение значений параметров в генерируемые журнальные сообщения в файле `Startup.cs` из папки `PartyInvites`

```
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.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using PartyInvites.Models;

namespace PartyInvites {

  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<DataContext>(options => {
       options.EnableSensitiveDataLogging(true);
       options.UseSqlServer(conString);
     });
    }
```

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

С помощью метода `EnableSensitiveDataLogging()` инфраструктуре Entity Framework Core указывается на необходимость включения значений параметров в журнальные сообщения. Перезапустите приложение, используя `dotnet`, и перейдите по ссылке `http://localhost:5000/home/listresponses`. Вы заметите, что инфраструктура Entity Framework Core включила в журнальное сообщение, сгенерированное приложением, значение параметра, которое отображается непосредственно перед запросом:

```
...
Executed DbCommand (58ms) [Parameters=[@__searchTerm_0='555-123-5678'
  (Size = 4000)], CommandType='Text', CommandTimeout='30']
SELECT [r].[Id], [r].[Email], [r].[Name], [r].[Phone], [r].[WillAttend]
FROM [Responses] AS [r]
WHERE [r].[Phone] = @__searchTerm_0
ORDER BY [r].[Email]
...
```

Сохранение и обновление данных

В большинстве приложений запросы являются самым распространенным типом команд, но процедуры сохранения новых и обновления существующих данных также важны. Команда `INSERT` применяется для вставки новой строки данных в таблицу; SQL-команда, показанная в листинге 3.10, добавляет новую строку в таблицу `Responses`.

Листинг 3.10. Вставка данных в таблицу

```
USE PartyInvites

INSERT INTO Responses(Name, Email, Phone, WillAttend)
VALUES ('Joe Dobbs', 'joe@example.com', '555-888-1234', 1)
```

Ключевые слова `INSERT INTO`, за которыми находится список столбцов, требующих предоставления значений. Затем используется ключевое слово `VALUES`, за которым следуют значения, подлежащие сохранению в БД, в том же самом порядке, что и столбцы в списке. Команда в листинге 3.10 добавляет новую строку со значениями для столбцов `Name`, `Email`, `Phone` и `WillAttend`. Значение для столбца `Id` не требуется, поскольку Entity Framework Core конфигурирует его так, что сервер баз данных сгенерирует значение `Id` при сохранении остальных столбцов.

Совет. Обратите внимание, что строки в SQL обозначаются с помощью одинарных кавычек (символ '), а не двойных (символ ") как в C#.

В результате выполнения команды в листинге 3.10 появится следующее сообщение:

```
...
(1 row(s) affected)
(1 строк(а) затронуто)
...
```

Сервер баз данных реагирует на команды, модифицирующие БД, указанием количества строк, которые были изменены. Чтобы увидеть команду INSERT, применяемую инфраструктурой Entity Framework Core, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке RSVP Now и создайте ответ с использованием значений из табл. 3.7.

Таблица 3.7. Значения для создания ответа

Name	Email	Phone	WillAttend
Anna Roth	anna@example.com	555-204-7692	False

Если вы просмотрите журнальные сообщения, сгенерированные приложением при сохранении новых данных в БД, то увидите, что применялась почти такая же элементарная команда INSERT, но с несколькими отличиями:

```
...
Executed DbCommand (3ms) [Parameters=[@p0='anna@example.com' (Size = 4000),
    @p1='Anna Roth' (Size = 4000), @p2='555-204-7692' (Size = 4000),
    @p3=False' (Nullable = true)], CommandType='Text', CommandTimeout='30']

SET NOCOUNT ON;

INSERT INTO [Responses] ([Email], [Name], [Phone], [WillAttend])
VALUES (@p0, @p1, @p2, @p3);
...
```

Команда SET NOCOUNT ON отключает сообщение о количестве строк, затронутых командой. Она малоэффективна для простых команд вроде тех, что приведена выше, но способна улучшить показатели производительности в случае более сложных операций. Как видите, команда INSERT, используемая Entity Framework Core, похожа на команду из листинга 3.10, но с параметризированными значениями.

Непосредственно после INSERT инфраструктура Entity Framework Core отправляет серверу баз данных еще одну команду:

```
...
SELECT [Id]
FROM [Responses]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

Эта команда применяется Entity Framework Core для проверки количества строк, затронутых командой INSERT, и для определения уникального значения, которое сервер баз данных присвоил столбцу Id. Знание значения Id важно для последующих операций, выполняемых над новыми сохраненными данными, и Entity Framework Core производит такой запрос, даже когда в приложении операции подобного рода отсутствуют.

Обновление существующих данных

Команда UPDATE используется для модификации данных, которые были сохранены в БД в прошлом, и может применяться для изменения множества строк в единственной операции. Команда, показанная в листинге 3.11, модифицирует строки в таблице Responses, чье значение WillAttend равно 1, и также изменяет их телефонный номер. (Само по себе это изменение не особенно полезно, однако оно демонстрирует базовую структуру UPDATE.)

Листинг 3.11. Пример обновления существующих данных

```
USE PartyInvites

UPDATE Responses
SET Phone='404-204-1234'
WHERE WillAttend = 1
```

За ключевым словом UPDATE следует имя таблицы, а ключевое слово SET используется для указания столбцов и новых значений. За ключевым словом WHERE находится выражение, которое выбирает строки, подлежащие изменению. На выполнение команды сервер баз данных отреагирует выдачей количества затронутых строк:

```
...
(5 row(s) affected)
(5 строк(а) затронуто)
...
```

Выполните команду, приведенную в листинге 3.12, чтобы запросить все строки в таблице Responses.

Листинг 3.12. Запрашивание всех строк

```
USE PartyInvites

SELECT * FROM Responses
```

В табл. 3.8 показаны результаты этого запроса и выделено полужирным влияние обновления, произведенного в листинге 3.11.

Таблица 3.8. Эффект от обновления существующих данных

Id	Email	Name	Phone	WillAttend
1	alice@example.com	Alice Jones	**404-204-1234**	1
2	bob@example.com	Bob Smith	555-123-1234	0
3	peter@example.com	Peter Davies	**404-204-1234**	1
4	dora@example.com	Dora Francis	**404-204-1234**	1
5	jane@example.com	Jane Marshall	**404-204-1234**	1
6	joe@example.com	Joe Dobbs	**404-204-1234**	1
7	anna@example.com	Anna Roth	555-204-7692	0

Удаление данных

Команда DELETE удаляет данные из БД. Такая команда должна применяться осторожно, поскольку она имеет конструкцию WHERE, которая позволяет в единственной команде выбрать множество строк, подлежащих удалению, и легко непредумышленно удалить из БД больше данных, чем планировалось. Команда в листинге 3.13 удаляет из таблицы Responses все строки, которые имеют нулевое значение WillAttend.

Листинг 3.13. Пример удаления данных

```
USE PartyInvites

DELETE FROM Responses
WHERE WillAttend = 0
```

Выполните эту команду и затем повторите запрос, показанный в листинге 3.12, для проверки, что отклоненные приглашения были удалены из БД. Поддержка удаления данных в текущем примере приложения отсутствует, но вы встретите такой тип команды далее в книге, в том числе в приложении SportsStore, разработка которого начнется в следующей главе.

Соединение данных

Базы данных могут содержать отношения между таблицами, которые инфраструктура Entity Framework Core использует для отслеживания взаимосвязей между объектами. Указанная возможность построена на основе описанных ранее команд INSERT и UPDATE, но полагается на более сложный тип запроса, известный как *соединение*, который объединяет данные из множества таблиц. Соединения могут сбивать с толку; для работы с инфраструктурой Entity Framework Core понимать соединения не обязательно, но полезно иметь о них общее представление.

На заметку! Не переживайте, если материал данного раздела станет понятным не сразу. Он обретет гораздо больший смысл после того, как вы прочитаете главу 14, включающую рассмотрение средства Entity Framework Core, которое применяет соединения, детально описанные в главах 15 и 16.

Подготовка базы данных

Чтобы продемонстрировать, как работает соединение, понадобится добавить в БД еще одну таблицу. Выполните команду, представленную в листинге 3.14, которая создаст таблицу по имени Preferences.

Листинг 3.14. Создание новой таблицы в БД

```
USE PartyInvites

DROP TABLE IF EXISTS Preferences

CREATE TABLE Preferences (
   Id bigint IDENTITY,
   Email nvarchar(max),
   NutAllergy bit,
   Teetotal bit,
   ResponseId bigint,
)
```

Команда DROP TABLE сообщает серверу баз данных о необходимости удаления таблицы Preferences, если она существует, что означает возможность многократного выполнения команд из листинга 3.14 без возникновения ошибки.

Команда CREATE TABLE используется для создания новой таблицы. Указывается имя таблицы и список столбцов с их типами, которые таблица будет содержать. Созданная в листинге 3.14 таблица имеет столбцы Id, Email, NutAllergy и Teetotal. Столбец Id был сконфигурирован с ключевым словом IDENTITY, которое делает сервер баз данных ответственным за создание уникальных значений и которое Entity Framework Core по умолчанию применяет к столбцам первичного ключа. Команда в листинге 3.15 вставляет новые строки в таблицы Responses и Preferences.

Листинг 3.15. Сохранение связанных данных

```
USE PartyInvites

INSERT INTO Responses(Name, Email, Phone, WillAttend)
VALUES ('Dave Habbs', 'dave@example.com', '555-777-1234', 1)

INSERT INTO Preferences (Email, NutAllergy, Teetotal)
VALUES ('dave@example.com', 0, 1)
```

Первая команда INSERT добавляет строку в таблицу Responses. Вторая команда INSERT добавляет строку в таблицу Preferences. Выполните команды из листинга 3.16, чтобы запросить упомянутые таблицы и увидеть новые данные.

Листинг 3.16. Запрашивание новых данных

```
USE PartyInvites

SELECT * FROM Responses
SELECT * FROM Preferences
```

Первая команда SELECT запрашивает из БД все строки Response и включает строку, показанную в табл. 3.9, которая была создана первой командой INSERT в листинге 3.15.

Таблица 3.9. Новая строка в таблице Responses

Id	Email	Name	Phone	WillAttend
8	dave@example.com	Dave Habbs	555-777-1234	1

Вторая команда SELECT запрашивает из БД все строки Preferences и производит результат, приведенный в табл. 3.10. Обратите внимание, что свойство ResponseId строки в таблице Preferences совпадает со значением Id для строки, показанной в табл. 3.9.

Таблица 3.10. Новая строка в таблице Preferences

Id	Email	NutAllergy	Teetotal
1	dave@example.com	0	1

Выполнение соединения

Существуют разные типы соединений, но в инфраструктуре Entity Framework Core используется так называемое *внутреннее соединение*, которое выбирает из таблиц строки, разделяющие общее значение. В листинге 3.17 представлен запрос, выполняющий внутреннее соединение на таблицах `Responses` и `Preferences`.

Листинг 3.17. Выполнение внутреннего соединения

```
USE PartyInvites

SELECT Responses.Email, Responses.Name, Preferences.NutAllergy,
Preferences.Teetotal
FROM Responses
INNER JOIN Preferences ON Responses.Email = Preferences.Email
```

Команда `SELECT` запрашивает у сервера баз данных столбцы из обеих таблиц, а ключевые слова `INNER JOIN` применяются для указания, что строки из таблицы `Responses` должны соединяться со строками из таблицы `Preferences`, когда они имеют одинаковые значения `Email`. В результате значения данных из обеих строк, созданных в листинге 3.16, объединяются, как показано в табл. 3.11.

Таблица 3.11. Результат соединения

Email	Name	NutAllergy	Teetotal
dave@example.com	Dave Habbs	0	1

Несмотря на простоту, пример соединения демонстрирует основополагающий механизм и помогает получить понятие о запросах, которые Entity Framework Core использует для извлечения данных из БД, хранящих сложные модели данных.

Резюме

В главе объяснялось, каким образом применять среду Visual Studio для инспектирования БД, которая хранит данные приложения. Также было показано, как выполнять элементарные SQL-команды, чтобы вы имели представление об особенностях использования БД инфраструктурой Entity Framework Core и могли проверить, что получаете ожидаемые результаты. В следующей главе будет начат процесс создания более сложного и реалистичного приложения SportsStore.

ГЛАВА 4

SportsStore: реальное приложение работы с данными

В этой главе начинается процесс построения более реалистичного проекта, который продемонстрирует совместное использование ASP.NET Core MVC и Entity Framework Core. Проект будет простым, но близким к реальности, и фокусироваться на часто применяемых средствах Entity Framework Core. Приложение представляет собой вариацию приложения SportsStore, которое я использовал во многих своих книгах, с концентрацией на данных и хранилище данных.

В текущей главе создается простое самодостаточное приложение ASP.NET Core MVC. В следующей главе в него добавляется поддержка инфраструктуры Entity Framework Core и сохранения данных приложения в БД. В дальнейших главах будут добавлены дополнительные операции над данными, расширена модель данных, обеспечена поддержка средств для покупателей и показано, как масштабировать приложение. Повсюду в главах, где строится приложение SportsStore, приводятся ссылки на главы, в которых основные средства описаны обособленно и более подробно. Начиная с главы 5, где в проект добавляется инфраструктура Entity Framework Core, также будут рассматриваться наиболее распространенные проблемы, с которыми возможно вы столкнетесь, и приведены объяснения, как их решать.

На заметку! Основное внимание в настоящей книге сосредоточено на инфраструктуре Entity Framework Core и на том, как ее можно применять в приложении MVC. Определенные аспекты приложения SportsStore вроде аутентификации для доступа к административным функциям опущены, тогда как другие, скажем, модель данных — расширены.

Создание проекта

Чтобы создать проект SportsStore, запустите Visual Studio и выберите в меню File (Файл) пункт New⇨Project (Создать⇨Проект). Укажите шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), введите SportsStore в поле Name (Имя) и щелкните на кнопке Browse (Обзор) для выбора подходящего места хранения проекта (рис. 4.1).

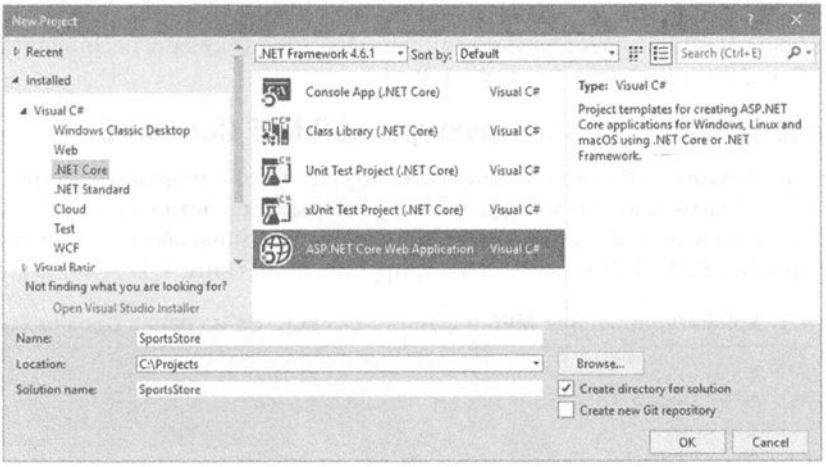

Рис. 4.1. Создание проекта для примера

Совет. Проект доступен для загрузки в хранилище GitHub по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Щелкните на кнопке OK, чтобы продолжить настройку проекта. 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. 4.2. Среда Visual Studio включает шаблоны, которые настраивают инфраструктуры ASP.NET Core MVC и Entity Framework Core в проекте, но результат скрывает ряд полезных деталей. Мы начнем с элементарного проекта ASP.NET Core и будем его пошагово строить в последующих главах, выясняя, каким образом различные компоненты взаимодействуют друг с другом.

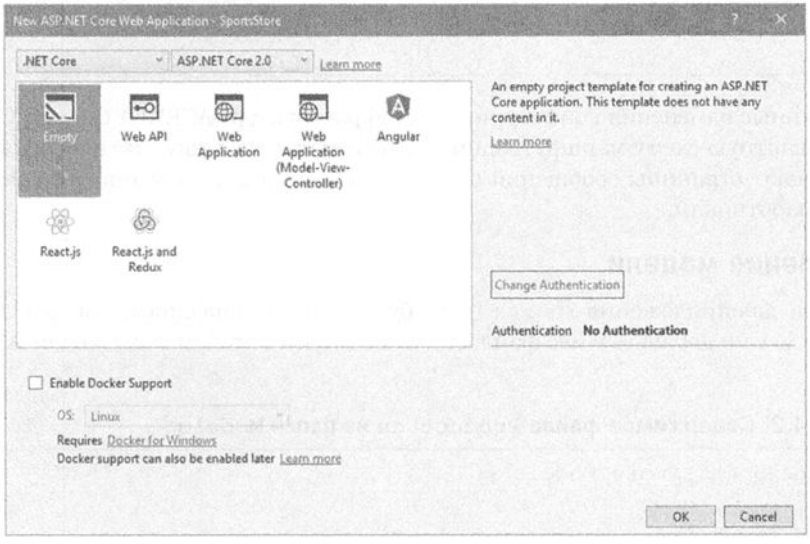

Рис. 4.2. Конфигурирование проекта ASP.NET Core

Щелкните на кнопке OK; среда Visual Studio создаст проект SportsStore с базовой конфигурацией, которая настраивает ASP.NET Core, но не инфраструктуру ASP.NET Core MVC или Entity Framework Core.

Конфигурирование инфраструктуры ASP.NET Core MVC

Далее необходимо добавить базовую конфигурацию для инфраструктуры ASP.NET Core MVC, чтобы можно было создать контроллеры и представления с целью обработки HTTP-запросов. Добавьте промежуточное программное обеспечение MVC в конвейер запросов ASP.NET Core, как иллюстрируется в листинге 4.1.

Листинг 4.1. Конфигурирование MVC в файле Startup.cs из папки SportsStore

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

namespace SportsStore {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
    }

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

Внесенные изменения конфигурируют инфраструктуру ASP.NET Core MVC, используя стандартную схему маршрутизации, добавляют поддержку статических файлов и настраивают страницы сообщений об ошибках для предоставления сведений, полезных разработчикам.

Добавление модели

Модель для приложения SportsStore будет основана на списке товаров. Создайте папку Models и добавьте в нее файл класса по имени Product.cs с кодом из листинга 4.2.

Листинг 4.2. Содержимое файла Product.cs из папки Models

```
namespace SportsStore.Models {

  public class Product {
    public string Name { get; set; }
    public string Category { get; set; }
```

```
    public decimal PurchasePrice { get; set; }
    public decimal RetailPrice { get; set; }
  }
}
```

Он является вариацией класса `Product`, который я обычно применяю в своих книгах, и позволит лучше продемонстрировать полезные функциональные средства Entity Framework Core.

Добавление хранилища

Мне нравится обеспечивать согласованный доступ к данным в приложении с использованием паттерна "Хранилище" (Repository), в котором интерфейс определяет свойства и методы, предназначенные для доступа к данным, а для работы с механизмом хранения данных применяется класс реализации. Преимущество использования паттерна "Хранилище" связано с облегчением модульного тестирования части MVC приложения, а также с тем, что детали, касающиеся хранения данных, скрыты от остальных частей приложения.

Совет. Применение хранилища — хорошая идея во многих проектах, но это не является требованием для работы с Entity Framework Core. Например, в части III книги большинство проектов не содержат хранилище из-за того, что иначе пришлось бы вносить в код большой объем сложных изменений и повторять их в многочисленных классах и интерфейсах. Не беспокойтесь, если вы не уверены в преимуществах паттерна "Хранилище", потому что вы всегда можете добавить хранилище позже, хотя и за счет некоторого рефакторинга.

Чтобы создать интерфейс хранилища, добавьте в папку `Models` файл по имени `IRepository.cs` и поместите в него код, приведенный в листинге 4.3.

Листинг 4.3. Содержимое файла `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public interface IRepository {

    IEnumerable<Product> Products { get; }

    void AddProduct(Product product);
  }
}
```

Свойство `Products` будет предоставлять доступ только по чтению ко всем товарам, известным приложению. Метод `AddProduct()` будет использоваться для добавления новых товаров.

В текущей главе мы собираемся хранить объекты модели в памяти, а в главе 5 заменим это инфраструктурой Entity Framework Core. Добавьте в папку `Models` файл класса по имени `DataRepository.cs` с кодом, показанным в листинге 4.4.

Листинг 4.4. Содержимое файла `DataRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {
```

```
  public class DataRepository : IRepository {
    private List<Product> data = new List<Product>();

    public IEnumerable<Product> Products => data;

    public void AddProduct(Product product) {
      this.data.Add(product);
    }
  }
}
```

Класс `DataRepository` реализует интерфейс `IRepository` и применяет экземпляр `List` для отслеживания объектов `Product`, т.е. в случае останова или перезапуска приложения данные будут утрачиваться. Постоянное хранилище будет введено в главе 5, а хранилища в памяти вполне достаточно для приведения в работоспособное состояние части ASP.NET Core MVC проекта, прежде чем добавлять часть Entity Framework Core.

Добавьте в класс `Startup` оператор, выделенный в листинге 4.5, чтобы зарегистрировать класс `DataRepository` как реализацию для использования в качестве зависимостей интерфейса `IRepository`.

Листинг 4.5. Конфигурирование внедрения зависимостей в файле `Startup.cs` из папки `SportsStore`

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

namespace SportsStore {
  public class Startup {

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddSingleton<IRepository, DataRepository>();
    }

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

Выделенный полужирным оператор в листинге 4.5 регистрирует класс `DataRepository` с применением метода `AddSingleton()`. В результате при распознавании зависимости интерфейса `IRepository` в первый раз создается одиночный объект, который будет использоваться для всех последующих зависимостей.

Добавление контроллера и представления

Основной акцент в рассматриваемом примере приложения сделан на управлении объектами товаров, потому что это создает великолепную возможность для демонстрации различных средств работы с данными. Нам нужен контроллер, который будет получать HTTP-запросы и транслировать их в операции над объектами `Product`, так что создайте папку `Controllers`, добавьте в нее файл по имени `HomeController.cs` и определите контроллер, как показано в листинге 4.6.

Листинг 4.6. Содержимое файла `HomeController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) => repository = repo;

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

    [HttpPost]
    public IActionResult AddProduct(Product product) {
      repository.AddProduct(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Метод действия `Index()` передает коллекцию объектов `Product` из хранилища своему представлению, которое отобразит пользователю экранную таблицу с данными. Метод `AddProduct()` сохраняет новые объекты `Product`, которые основаны на данных, полученных в HTTP-запросе `POST`.

Далее создайте папку `Views/Home` и поместите в нее файл по имени `Index.cshtml` с содержимым из листинга 4.7. Он является представлением, которое будет отображать данные `Product` приложения и разрешать пользователю создавать новые объекты.

Листинг 4.7. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  <form asp-action="AddProduct" method="post">
    <div class="row">
```

```
      <div class="col"><input name="Name" class="form-control" /></div>
      <div class="col"><input name="Category" class="form-control" /></div>
      <div class="col">
        <input name="PurchasePrice" class="form-control" />
      </div>
      <div class="col">
        <input name="RetailPrice" class="form-control" />
      </div>
      <div class="col">
        <button type="submit" class="btn btn-primary">Add</button>
      </div>
    </div>
  </form>
  @if (Model.Count() == 0) {
    <div class="row">
      <div class="col text-center p-2">No Data</div>
    </div>
  } else {
    @foreach (Product p in Model) {
      <div class="row p-2">
        <div class="col">@p.Name</div>
        <div class="col">@p.Category</div>
        <div class="col text-right">@p.PurchasePrice</div>
        <div class="col text-right">@p.RetailPrice</div>
        <div class="col"></div>
      </div>
    }
  }
</div>
```

Сеточная компоновка применяется для отображения встроенной формы, с помощью которой создаются новые объекты, наряду с деталями всех известных приложению объектов `Product` или заполнителем, если объекты отсутствуют.

Добавление последних штрихов

Щелкните правой кнопкой мыши на элементе проекта `SportsStore` в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон JSON File (Файл JSON), находящийся в категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), чтобы создать файл `.bowerrc` с содержимым, приведенным в листинге 4.8. (Важно обратить внимание на имя файла: оно начинается с точки, содержит две буквы `r` и не имеет расширения.)

Листинг 4.8. Содержимое файла `.bowerrc` из папки `SportsStore`

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

Снова воспользуйтесь шаблоном JSON File и создайте файл по имени `bower.json`, содержимое которого показано в листинге 4.9.

Листинг 4.9. Содержимое файла `bower.json` из папки `SportsStore`

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

Когда вы сохраните изменения в файле, среда Visual Studio загрузит новую версию пакета Bootstrap и установит его в папку `wwwroot/lib`.

На заметку! Инструмент управления пакетами клиентской стороны Bower, который применяется в этой главе для установки Bootstrap, объявлен устаревшим и со временем будет заменен. Причина его использования в том, что поддержка Bower остается активной и в Visual Studio есть встроенная поддержка для его применения.

Создайте папку `Views/Shared`, добавьте в нее файл по имени `_Layout.cshtml` и используйте его для определения разделяемой компоновки, приведенной в листинге 4.10.

Листинг 4.10. Содержимое файла `_Layout.cshtml` из папки `Views/Shared`

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

Простая компоновка `_Layout.cshtml` предоставляет структуру HTML-документа, требуемую браузерами, так что ее не придется включать в каждое представление. Она также содержит элемент `link`, сообщающий браузерам о необходимости запроса CSS-файла со стилями Bootstrap, которые применяются для стилизации содержимого повсеместно в книге.

Чтобы компоновка из листинга 4.10, использовалась по умолчанию, добавим в папку `Views` файл по имени `_ViewStart.cshtml` с содержимым из листинга 4.11. (В случае если для создания файла `_ViewStart.cshtml` применяется шаблон элемента MVC View Start Page (Страница запуска представления MVC), то среда Visual Studio добавит содержимое, показанное в листинге 4.11, автоматически.)

Листинг 4.11. Содержимое файла `_ViewStart.cshtml` из папки `Views`

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

Для включения вспомогательных функций дескрипторов ASP.NET Core MVC и облегчения ссылки на класс модели добавьте в папку `Views` файл по имени `_ViewImports.cshtml` и поместите в него содержимое из листинга 4.12.

Листинг 4.12. Содержимое файла `_ViewImports.cshtml` из папки `Views`

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

Финальный шаг заключается в настройке приложения для прослушивания порта 5000 на предмет поступления HTTP-запросов. Отредактируйте файл `launchSettings.json` в папке `Properties`, заменив произвольно назначенный порт, как продемонстрировано в листинге 4.13. Никакого особого значения с портом 5000 не связано, кроме того, что он используются во всех примерах, рассматриваемых в книге.

Листинг 4.13. Изменение порта 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"
      }
    },
    "SportsStore": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Запуск примера приложения

Чтобы скомпилировать и запустить пример приложения, откройте окно командной строки или PowerShell, перейдите в папку проекта `SportsStore` (ту, что содержит файл `bower.json`) и введите команду, приведенную в листинге 4.14.

Листинг 4.14. Запуск на выполнение примера приложения

```
dotnet run
```

Приложение начнет прослушивать порт 5000 на предмет поступления HTTP-запросов. Откройте окно браузера и перейдите по ссылке `http://localhost:5000`. Вы увидите первоначальный заполнитель, который будет заменен после того, как вы введете данные в полях формы и щелкнете на кнопке Add (Добавить), как показано на рис. 4.3.

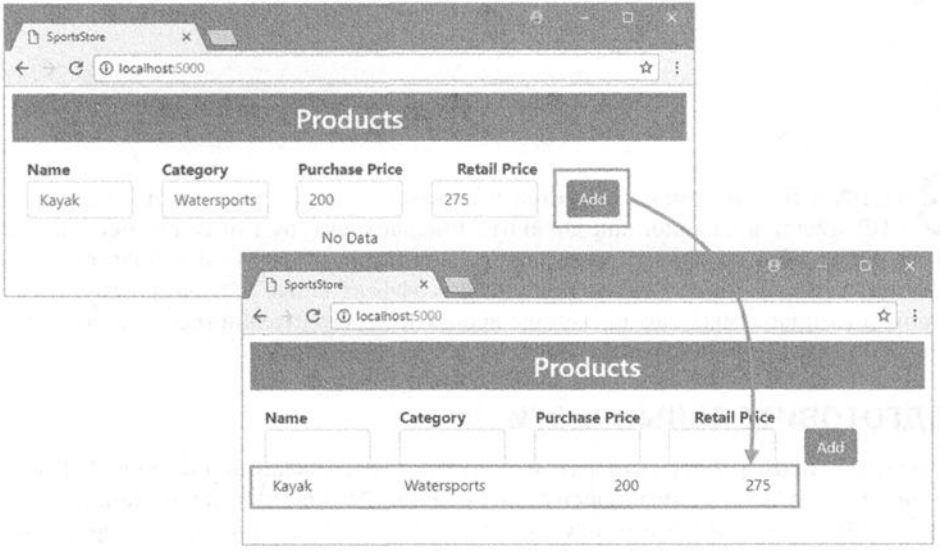

Рис. 4.3. Выполнение примера приложения

Резюме

В главе было создано простое приложение ASP.NET Core MVC, которое планируется применять в последующих главах. Сейчас данные приложения хранятся в памяти, т.е. в случае останова или перезапуска приложения сведения обо всех товарах утрачиваются. В следующей главе в проект будет добавлена поддержка Entity Framework Core для обеспечения постоянного хранения данных в БД.

ГЛАВА 5

SportsStore: хранение данных

В настоящей главе демонстрируется сохранение данных приложения SportsStore в БД. Будет показано, как добавить инфраструктуру Entity Framework Core в проект, каким образом подготовить модель данных, как создать и использовать БД и каким образом заставить приложение делать эффективные SQL-запросы. Также будут описаны проблемы, с которыми вы вероятнее всего столкнетесь при добавлении Entity Framework Core в проект, и предложены их решения.

Подготовительные шаги

Мы продолжим работу с проектом SportsStore, созданным в главе 4. Никаких изменений для этой главы вносить не нужно. Откройте окно командной строки PowerShell, перейдите в папку проекта SportsStore (ту, что содержит файл bower.json) и с помощью команды dotnet run запустите приложение. В окне браузера перейдите по ссылке http://localhost:5000; появится содержимое, представленное на рис. 5.1. Вы можете применять HTML-форму для сохранения объектов Product, но в случае останова или перезапуска приложения они будут утрачены, потому что данных хранятся только в памяти.

Рис. 5.1. Выполнение примера приложения

Совет. Проект SportsStore и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Конфигурирование инфраструктуры Entity Framework Core

Стандартная конфигурация, которую Visual Studio создает для проектов ASP.NET Core, включает пакеты NuGet, требующиеся для запуска приложений Entity Framework Core. Необходим еще отдельный пакет для добавления инструментов командной строки, которые управляют базами данных, и он должен устанавливаться вручную.

Щелкните правой кнопкой мыши на элементе проекта SportsStore в окне Solution Explorer, выберите в контекстном меню пункт Edit SportsStore.csproj (Редактировать SportsStore.csproj) и добавьте элемент конфигурации, приведенный в листинге 5.1.

Листинг 5.1. Добавление пакета в файле SportsStore.csproj из папки SportsStore

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

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

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
    <DotNetCliToolReference
       Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
       Version="2.0.0" />
  </ItemGroup>

</Project>
```

Пакеты, включающие инструменты командной строки, должны добавляться вручную с использованием элемента DotNetCliToolReference. Указанный в листинге 5.1 пакет содержит команды dotnet ef, которые применяются для управления базами данных в проектах, использующих Entity Framework Core.

Конфигурирование журнальных сообщений Entity Framework Core

Даже в проекте, сохраняющем лишь небольшой объем данных, важно понимать запросы и команды SQL, которые инфраструктура Entity Framework Core посылает серверу баз данных. Чтобы настроить Entity Framework Core на генерацию журнальных сообщений, которые будут отображать применяемые SQL-запросы, добавьте в папку SportsStore файл по имени appsettings.json с использованием шаблона элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) и поместите в него элементы конфигурации из листинга 5.2.

Листинг 5.2. Содержимое файла appsettings.json из папки SportsStore

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

Стандартное содержимое, применяемое средой Visual Studio для этого файла, включает строку подключения к БД, которая вскоре будет изменена. Выделенный полужирным код в листинге 5.2 устанавливает уровень ведения журнала по умолчанию в None, что запрещает все журнальные сообщения. Затем установка переопределяется для пакета Microsoft.EntityFrameworkCore с указанием настройки Information, которая обеспечит предоставление деталей SQL-кода, используемого Entity Framework Core. В реальных проектах вы не должны отключать все остальные журнальные сообщения, но такая комбинация облегчит отслеживание примеров.

Подготовка модели данных

В приводимых ниже разделах модель данных, уже существующая в проекте SportsStore, будет подготовлена для применения с инфраструктурой Entity Framework Core.

Определение свойства первичного ключа

Для сохранения данных в БД инфраструктура Entity Framework Core должна уметь уникальным образом идентифицировать каждый объект, что требует выбора свойства, которое будет использоваться как первичный ключ. В большинстве проектов простейший способ определения первичного ключа предусматривает добавление в класс модели данных свойства типа long по имени Id (листинг 5.3).

Листинг 5.3. Добавление свойства первичного ключа в файле Product.cs из папки Models

```
namespace SportsStore.Models {

  public class Product {

    public long Id { get; set; }

    public string Name { get; set; }
    public string Category { get; set; }
    public decimal PurchasePrice { get; set; }
    public decimal RetailPrice { get; set; }
  }
}
```

Этот подход означает, что инфраструктура Entity Framework Core будет конфигурировать БД так, чтобы сервер баз данных генерировал значения первичного ключа, и вам не пришлось беспокоиться об избегании дубликатов. Применение значения long гарантирует наличие большого диапазона значений первичного ключа, и большинство проектов будут в состоянии неограниченно хранить данные, не тревожась о том, что ключи закончатся.

Создание класса контекста базы данных

В обеспечении доступа к данным в БД инфраструктура Entity Framework Core полагается на класс контекста БД. Чтобы снабдить пример приложения контекстом, добавьте в папку Models файл класса по имени DataContext.cs с кодом из листинга 5.4.

Листинг 5.4. Содержимое файла DataContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
  public class DataContext : DbContext {
    public DataContext(DbContextOptions<DataContext> opts) : base(opts)
{}
    public DbSet<Product> Products { get; set; }
  }
}
```

Когда инфраструктура Entity Framework Core используется для сохранения простой модели данных вроде той, что определена в приложении SportsStore, класс контекста БД соответственно прост — хотя ситуация изменится с ростом сложности модели данных в последующих главах. Пока что класс контекста БД обладает тремя важными характеристиками.

Первая характеристика касается того, что базовым классом является DbContext, который определен в пространстве имен Microsoft.EntityFrameworkCore. Именно применение DbContext в качестве базового класса создает контекст БД и обеспечивает доступ к функциональности Entity Framework Core.

Вторая характеристика заключается в том, что конструктор получает объект DbContextOptions<T> (где T — класс контекста), который должен быть передан конструктору базового класса с использованием ключевого слова base:

```
...
public DataContext(DbContextOptions<DataContext> opts) : base(opts) { }
...
```

Параметр конструктора предоставит инфраструктуре Entity Framework Core конфигурационную информацию, необходимую для подключения к серверу баз данных. Вы получите ошибку, если не определите параметр конструктора или не передадите объект.

Третья характеристика связана со свойством типа DbSet<T>, где T — класс, который планируется сохранять в БД:

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

Классом модели данных является Product, поэтому свойство в листинге 5.4 возвращает объект DbSet<Product>. Свойство должно определяться с конструкциями get и set. Конструкция set позволяет инфраструктуре Entity Framework Core присваивать объект, который обеспечивает удобный доступ к данным. Конструкция get предоставляет доступ к таким данным остальным частям приложения.

Обновление реализации хранилища

Теперь пора обновить класс реализации хранилища, чтобы обращаться к данным через класс контекста, определенный в предыдущем разделе (листинг 5.5).

Листинг 5.5. Применение класса контекста в файле DataRepository.cs из папки Models

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    // private List<Product> data = new List<Product>();
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

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

    public void AddProduct(Product product) {
      this.context.Products.Add(product);
      this.context.SaveChanges();
    }
  }
}
```

В приложении ASP.NET Core MVC доступ к объектам контекста данных управляется с использованием внедрения зависимостей и потому в класс DataRepository был добавлен конструктор, принимающий объект DataContext, который будет предоставлен средством внедрения зависимостей во время выполнения.

Свойство Products, определенное интерфейсом хранилища, может быть реализовано путем возвращения свойства DbSet<Product>, которое определено в классе контекста. Аналогично метод AddProduct() реализовать легко, потому что объект DbSet<Product> определяет метод Add(), который принимает объекты Product и сохраняет их на постоянной основе.

Самым важным изменением является добавление вызова метода SaveChanges(), сообщающего инфраструктуре Entity Framework Core об отправке в БД любых ожидающих операций наподобие запросов к методу Add() для сохранения данных.

Подготовка базы данных

В последующих разделах мы пройдем через процесс конфигурирования приложения SportsStore, чтобы описать применяемую БД и затем запросить у Entity Framework Core ее создание. Такой подход известен как проект "сначала код", когда вы начинаете с одного или большего числа классов C# и используете их для создания и конфигурирования БД. Альтернативный подход называется проектом "сначала БД", в котором вы создаете модель данных из существующей БД — процесс, описываемый в главах 17 и 18.

Конфигурирование строки подключения

При предоставлении сведений о том, как связываться с сервером баз данных, который применяет класс контекста, инфраструктура Entity Framework Core полагается на строки подключения. Формат строк подключения варьируется в зависимости от используемого сервера баз данных, но обычно предполагает указание имени сервера и сетевого порта на сервере баз данных, имени БД и учетных данных аутентификации.

Строки подключения определяются в файле `appsettings.json`, а в листинге 5.6 представлено определение строки подключения для БД приложения `SportsStore`. В книге применяется версия LocalDB продукта SQL Server, которая спроектирована специально для разработчиков и не требует конфигурирования или учетных данных.

Совет. Вы обязаны обеспечить, чтобы строка подключения была единственной неразрывной строкой. На печатной странице это затруднено, но вы получите ошибку, если ради улучшения читабельности разобьете строку подключения на несколько строк.

Формат строки подключения специфичен для каждого сервера баз данных. В строке подключения из листинга 5.6 есть четыре конфигурационных свойства, которые описаны в табл. 5.1.

Листинг 5.6. Добавление строки подключения в файле `appsettings.json` из папки `SportsStore`

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

Конфигурирование поставщика базы данных и класса контекста

Добавьте в файл `Startup` операторы конфигурации, показанные в листинге 5.7, чтобы сообщить инфраструктуре Entity Framework Core о том, каким образом использовать строку подключения, которую должен применять поставщик БД, и как управлять классом контекста.

Таблица 5.1. Четыре конфигурационных свойства строки подключения LocalDB

Имя	Описание
`Server`	Указывает имя сервера баз данных, к которому будет подключаться инфраструктура Entity Framework Core. Для LocalDB значением является `(localdb)\\MSSQLLocalDB`, что делает возможным подключение к серверу баз данных, не требуя какой-то дополнительной конфигурации
`Database`	Указывает имя БД, которую будет использовать инфраструктура Entity Framework Core. В листинге 5.6 удален заполнитель, добавляемый в файл средой Visual Studio, и в качестве имени указано `SportsStore`
`Trusted_Connection`	Когда установлено в `true`, инфраструктура Entity Framework Core будет выполнять аутентификацию на сервере баз данных с применением учетной записи Windows. В случае LocalDB это свойство не требуется, хотя среда Visual Studio по умолчанию добавляет его при создании файла `appsettings.json`
`MultipleActiveResultSets`	Конфигурирует подключение к серверу баз данных так, чтобы одновременно можно было читать результаты из множества запросов

Листинг 5.7. Конфигурирование Entity Framework Core 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.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace SportsStore {
  public class Startup {

    public Startup(IConfiguration config) => Configuration = config;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddTransient<IRepository, DataRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
        options.UseSqlServer(conString));
    }
```

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

Конструктор и свойство `Configuration` используются для доступа к данным конфигурации в файле `appsettings.json`, что позволяет прочитать строку подключения. Расширяющий метод `AddDbContext<T>()` применяется для настройки класса контекста, указывает инфраструктуре Entity Framework Core, какой поставщик БД использовать (в этом случае посредством метода `UseSqlServer()`, но для каждого поставщика БД предусмотрен свой метод), и предоставляет строку подключения.

Обратите внимание, что изменен также и метод, который конфигурирует внедрение зависимостей для интерфейса `IRepository`:

```
...
services.AddTransient<IRepository, DataRepository>();
...
```

В главе 4 с помощью метода `AddSingleton()` обеспечивалось применение одиночного объекта `DataRepository` для распознавания всех зависимостей интерфейса `IRepository`, что было важно, поскольку данные приложения хранились в экземпляре `List`, и желательно было всегда использовать тот же самый объект. Теперь, когда применяется инфраструктура Entity Framework Core, задействован метод `AddTransient()`, который гарантирует создание нового объекта `DataRepository` при каждом распознавании зависимости интерфейса `IRepository`. Такой подход важен, потому что инфраструктура Entity Framework Core рассчитывает на создание нового объекта контекста для каждого HTTP-запроса в приложении ASP.NET Core MVC.

Создание базы данных

В предыдущем разделе для инфраструктуры Entity Framework Core был указан вид данных, подлежащих хранению, и способ подключения к серверу баз данных. Далее необходимо создать БД.

Инфраструктура Entity Framework Core управляет базами данных через средство, называемое *миграциями*, которые представляют собой наборы изменений, создающих или модифицирующих БД с целью ее синхронизации с моделью данных (миграции будут подробно описаны в главе 13). Чтобы создать миграцию, которая настроит БД, откройте окно командной строки или PowerShell, перейдите в папку проекта `SportsStore` (папку, содержащую файл `bower.json`) и выполните команду, приведенную в листинге 5.8.

Листинг 5.8. Создание миграции

```
dotnet ef migrations add Initial
```

Команды `dotnet ef` обращаются к функциональным средствам из пакета, добавленного в листинге 5.1. Аргументы `migrations add` сообщают инфраструктуре Entity

Framework Core о необходимости создания новой миграции, а последний аргумент указывает имя миграции, `Initial` — общепринятое имя, используемое для миграции, которая производит первоначальную подготовку БД.

При выполнении команды из листинга 5.8 инфраструктура Entity Framework Core инспектирует проект, находит класс контекста и применяет его для создания миграции. В результате в окне Solution Explorer появится папка `Migrations`, содержащая файлы классов, операторы которых подготовят БД.

Просто создать миграцию, которая представляет собой всего лишь набор инструкций, недостаточно. Инструкции миграции должны быть выполнены, чтобы создать БД, которая сможет хранить данные приложения. Для выполнения инструкций в миграции `Initial` запустите в папке проекта `SportsStore` команду, показанную в листинге 5.9.

Совет. Если вы прорабатывали примеры в этой главе и видите сообщение об ошибке, указывающее о том, что объект по имени `Products` уже существует, тогда прежде чем запускать команды из листинга 5.9, выполните в папке проекта команду `dotnet ef database drop --force`, чтобы удалить базу данных.

Листинг 5.9. Применение миграции

```
dotnet ef database update
```

Инфраструктура Entity Framework Core подключится к серверу баз данных, указанному в строке подключения, и выполнит операторы в миграции. Результатом будет БД, которую можно использовать для хранения объектов `Product`.

Выполнение приложения

Основная поддержка для постоянного хранения объектов `Product` на месте и приложение готово к тестированию, несмотря на то, что работа над ним еще не завершена. Запустите приложение, введя команду `dotnet run` в папке проекта `SportsStore`, перейдите по ссылке `http://localhost:5000` и с помощью HTML-формы создайте объекты `Product`, применяя значения из табл. 5.2.

Таблица 5.2. Значения для создания тестовых объектов `Product`

Name (Наименование)	Category (Категория)	Purchase Price (Покупная цена)	Retail Price (Розничная цена)
Kayak (Каяк)	Watersports (Водный спорт)	200	275
Lifejacket (Спасательный жилет)	Watersports (Водный спорт)	30	48.95
Soccer Ball (Футбольный мяч)	Soccer (Футбол)	17	19.50

После ввода каждого набора значений данных щелкайте на кнопке Add (Добавить) и Entity Framework Core сохранит объект в БД, приводя в итоге к результату, который представлен на рис. 5.2.

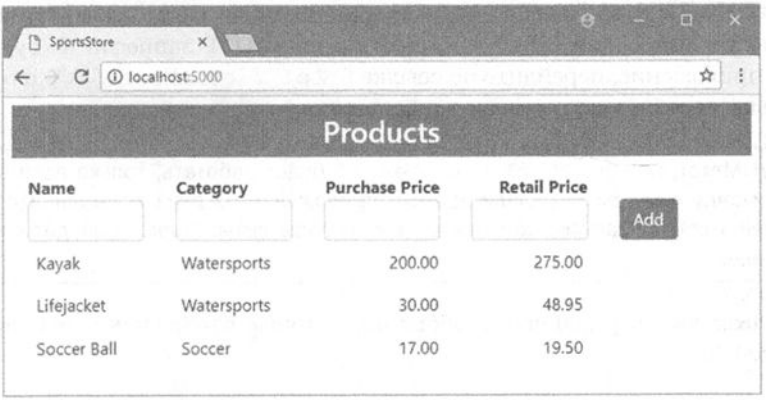

Рис. 5.2. Тестирование хранилища данных

Пользовательский интерфейс остался неизменным, но внутренне инфраструктура Entity Framework Core сохраняет данные в БД. Остановите и перезапустите приложение, используя команду `dotnet run`; вы увидите, что введенные ранее данные по-прежнему доступны.

Избегание ловушек, связанных с запросами

Приложение функционирует, данные сохраняются в БД, но все еще есть работа, которую нужно сделать, чтобы извлечь максимум из Entity Framework Core. В частности, необходимо избежать двух распространенных ловушек. Такие проблемы можно идентифицировать, исследуя SQL-запросы, которые Entity Framework Core посылает БД. С этой целью добавьте в метод действия `Index()` контроллера `Home` оператор, облегчающий просмотр запросов к БД, которые инициируются HTTP-запросом (листинг 5.10).

Листинг 5.10. Добавление оператора для работы с консолью в файле `HomeController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
  public class HomeController : Controller {
    private IRepository repository;
    public HomeController(IRepository repo) => repository = repo;
    public IActionResult Index() {
      System.Console.Clear();
      return View(repository.Products);
    }
    [HttpPost]
    public IActionResult AddProduct(Product product) {
      repository.AddProduct(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Метод `System.Console.Clear()` очищает консоль, когда вызывается действие `Index`, так что запросы к БД из предшествующих HTTP-запросов не будут видны. Запустите приложение, перейдите по ссылке `http://localhost:5000` и просмотрите журнальные сообщения, которые отобразились.

На заметку! Метод `System.Console.Clear()` будет работать, только если вы применяете команду `dotnet run` для запуска приложения из окна командной строки или PowerShell. Попытка запуска приложения в отладчике Visual Studio приведет к генерации исключения.

Вы увидите два журнальных сообщения, которые отображают два запроса, отправленные БД:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
...
```

В последующих разделах объясняется, почему запросов два и по какой причине один из них не использует в своих интересах все возможности сервера баз данных.

Ловушка, связанная с `IEnumerable<T>`

Инфраструктура Entity Framework Core облегчает запрашивание БД с применением LINQ, хотя и не всегда работает так, как можно было бы ожидать. В представлении `Index`, используемом контроллером `Home`, для выяснения количества объектов `Product`, которые были сохранены в БД, применяется метод `Count()` из LINQ:

```
...
@if (Model.Count() == 0) {
  <div class="row">
    <div class="col text-center p-2">No Data</div>
  </div>
} else {
  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col">@p.Name</div>
      <div class="col">@p.Category</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
      <div class="col"></div>
    </div>
  }
}
...
```

Чтобы определить, сколько объектов `Product` сохранилось в БД, инфраструктура Entity Framework Core использует SQL-оператор `SELECT` для получения всех доступных данных `Product`, применяет полученные данные для создания последователь-

ности объектов Product и затем подсчитывает их. После завершения подсчета объекты Product отбрасываются.

Когда в БД есть только три объекта, сложностей не возникает, но с ростом количества объектов объем работы, требуемой для их подсчета в такой манере, становится проблемой. Более эффективный подход предусматривает выполнение подсчета сервером баз данных, что освобождает инфраструктуру Entity Framework Core от необходимости перемещения всех данных и создания объектов. Реализовать такой подход можно, внеся в тип модели представления одно простое изменение, как показано в листинге 5.11.

Листинг 5.11. Изменение модели представления в файле Index.cshtml из папки Views/Home

```
@model IQueryable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <!-- ...для краткости остальные элементы не показаны... -->
</div>
```

Перезагрузив окно браузера, вы увидите, что первые два запроса, которые Entity Framework Core посылает серверу баз данных, изменились:

```
...
SELECT COUNT(*)
FROM [Products] AS [p]
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
...
```

Запрос SELECT COUNT предлагает серверу баз данных подсчитать количество объектов Product, не извлекая данные и не создавая любые объекты в приложении.

Получение разных запросов для разных типов моделей представлений может выглядеть как нелогичное поведение, и понимание причин, почему так происходит, жизненно важно в гарантировании инфраструктуре Entity Framework Core возможностей эффективного запрашивания баз данных.

Язык LINQ построен в виде набора расширяющих методов, которые оперируют на объектах, реализующих интерфейс IEnumerable<T>. Этот интерфейс представляет последовательность объектов и реализован обобщенными классами коллекций и массивов.

Инфраструктура Entity Framework Core включает дублирующий набор расширяющих методов LINQ, которые оперируют на объектах, реализующих интерфейс IQueryable<T>. Этот интерфейс представляет запрос к БД, и такое дублирование означает, что операции вроде Count() могут столь же легко выполняться над данными в БД, как над объектами в памяти.

Класс DbSet<T>, который использовался в классе контекста БД, созданном в листинге 5.4, реализует оба упомянутых интерфейса, так что, например, свойство Products реализует IEnumerable<Product> и IQueryable<T>. Когда модель представления в представлении Index была установлена в IEnumerable<Product>, применялась стандартная версия метода Count(). Стандартная реализация Count()

не имеет ни малейшего понятия об инфраструктуре Entity Framework Core и просто подсчитывает объекты в последовательности. В итоге инициируется запрос SELECT и возникает неэффективное поведение, приводящее к чтению всех данных для создания объектов, которые подсчитываются и отбрасываются.

После изменения модели представления на IQueryable<Product> использовалась версия метода Count() из Entity Framework Core, позволяющая инфраструктуре Entity Framework Core транслировать полный запрос в SQL и обеспечивать более эффективное поведение, при котором для получения количества сохраненных объектов применяется SELECT COUNT без необходимости в извлечении каких-либо данных.

Приведение модели представления Razor

Вас может удивить, что объект модели представления можно трактовать как объект IQueryable<Product>, хотя результатом свойства Products класса хранилища является IEnumerable<T>. Когда представления компилируются, класс C# включает явное преобразование в тип, указанный моделью представления, которое подобно включению следующей операции в метод действия:

```
...
public IActionResult Index() {
  System.Console.Clear();
  return View(repository.Products as IQueryable<Product>);
}
...
```

В рассмотренном примере такая характеристика означает возможность переключения работы с интерфейсами IQueryable<T> и IEnumerable<T> простым изменением выражения @model. Преобразование типов происходит во время выполнения, из-за чего любые несоответствия между объектом, предоставленным контроллером, и объектом, выраженным представлением, не будет видно вплоть до времени выполнения приложения.

Ловушка, связанная с дублированным запросом

Повышение эффективности одного из запросов не объясняет первопричину наличия двух запросов. Как объяснялось в предыдущем разделе, класс DbSet<T> реализует интерфейс IQueryable<T>, который представляет запрос к БД и позволяет использовать LINQ даже на данных в БД.

По умолчанию инфраструктура Entity Framework Core не выполняет запрос, пока не начнется перечисление объекта IQueryable<T>. Это дает возможность составлять запросы постепенно и создавать новые запросы, вызывая метод LINQ на существующем запросе, а не на возвращаемых им данных. Но указанное поведение также означает, что каждый раз, когда объект IQueryable<T> перечисляется, в БД посылается новый SQL-запрос. В некоторых приложениях такое поведение удобно, поскольку оно означает, что для получения самых последних данных из БД можно применять один и тот же объект. Однако в приложении ASP.NET Core MVC подобное поведение обычно приводит к выпуску множества запросов для тех же самых данных спустя всего несколько миллисекунд.

В примере приложения объект модели представления IQueryable<T> дважды перечисляется в представлении Index:

```
...
@if (Model.Count() == 0) {
  <div class="row">
    <div class="col text-center p-2">No Data</div>
  </div>
} else {
  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col">@p.Name</div>
      <div class="col">@p.Category</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
      <div class="col"></div>
    </div>
  }
}
...
```

Запрос инициируют не только циклы `foreach`, выполняющие перечисление последовательностей объектов; методы LINQ, которые производят одиночный результат, такие как `Count()`, также запускают запрос. Поведение `IQueryable<T>` и его использование в представлении `Index` вместе вырабатывают два запроса.

Теперь ситуация может показаться улучшенной, т.к. два запроса больше не идентичны, но возможны дальнейшие усовершенствования, которые будут описаны в последующих разделах.

Избегание непреднамеренных запросов

В инициировании множества запросов из одного объекта `IQueryable<T>` нет ничего плохого при условии, что именно в этом состояло ваше намерение. Проблема возникает, когда вы забываете, каким образом ведут себя объекты `IQueryable<T>`, обращаетесь с ними как с объектами `IEnumerable<T>` и непредумышленно делаете запросы, не замечая их. В высоконагруженном приложении объем ресурсов, впустую растрачиваемых на несущественные запросы, может стать значительным и привести к увеличению затрат в проекте.

Избегание запроса с использованием CSS

Представление `Index` демонстрирует одну из самых распространенных причин дублирования запросов в приложении ASP.NET Core MVC, где с применением метода `Count()` выясняется, есть ли какие-то данные, чтобы содержимое заполнителя можно было отобразить пользователю. Альтернативный способ предоставления заполнителя, сообщающего об отсутствии данных ("No Data"), предусматривает перенос этой ответственности на браузер, полагаясь на CSS. В листинге 5.12 в компоновку, используемую представлениями примера приложения, добавлен элемент `style`, который применяется для определения двух специальных стилей.

Листинг 5.12. Определение стилей в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
```

```
  <title>SportsStore</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <style>
    .placeholder { visibility: collapse; display: none }
    .placeholder:only-child { visibility: visible; display: flex }
  </style>
</head>
<body>
  <div class="p-2">
    @RenderBody()
  </div>
</body>
</html>
```

По умолчанию HTML-элемент, которому назначается класс `placeholder`, будет иметь свойство `visibility`, установленное в `collapse`, и свойство `display`, установленное в `none`, что не позволит пользователю его видеть. Но когда HTML-элемент является единственным дочерним элементом включающего элемента, значения свойств изменятся, что достигается за счет использования псевдокласса `only-child`. В листинге 5.13 приведено переработанное представление `Index`, применяемое контроллером `Home`, в котором вместо вызова метода `Count()` из LINQ используются классы CSS.

Листинг 5.13. Применение классов CSS в файле `Index.cshtml` из папки `Views/Home`

```
@model IQueryable<Product>
<h3 class="p-2 bg-primary text-white text-center">Products</h3>
<div class="container-fluid mt-3">
  <div class="row">
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  <form asp-action="AddProduct" method="post">
    <div class="row">
      <div class="col"><input name="Name" class="form-control" /></div>
      <div class="col"><input name="Category" class="form-control" /></div>
      <div class="col">
        <input name="PurchasePrice" class="form-control" />
      </div>
      <div class="col">
        <input name="RetailPrice" class="form-control" />
      </div>
      <div class="col">
        <button type="submit" class="btn btn-primary">Add</button>
      </div>
    </div>
  </form>
  <div>
    <div class="row placeholder">
      <div class="col text-center p-2">No Data</div>
    </div>
```

```
    @foreach (Product p in Model) {
      <div class="row p-2">
        <div class="col">@p.Name</div>
        <div class="col">@p.Category</div>
        <div class="col text-right">@p.PurchasePrice</div>
        <div class="col text-right">@p.RetailPrice</div>
        <div class="col"></div>
      </div>
    }
  </div>
</div>
```

Здесь добавляется элемент `div`, чтобы заработал псевдокласс `only-child`, и удалена конструкция `if` с вызовом метода `Count()`. В результате элемент, которому назначен класс `placeholder`, будет всегда включаться в HTML-разметку, отправляемую браузеру, но видимым окажется, только если цикл `foreach` не генерирует каких-либо элементов, что происходит, когда в БД нет ни одного объекта `Product`. Перезагрузив страницу, вы увидите, что теперь БД посылается единственный запрос:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
...
```

Принудительное выполнение запросов в хранилище

Проблема при работе с объектами `IQueryable<T>` напрямую связана с тем, что детали реализации хранилища данных просачиваются в другие части приложения, сводя на нет смысл функционального разделения, которому следует паттерн MVC.

Принятие сбалансированного подхода в отношении паттернов

Паттерны являются удобными шаблонами для разработки проектов, которые просто понимать и легко тестировать, но когда дело доходит до их реализации, требуется сбалансированный подход. Не имеет значения, какие части паттерна вы адаптируете, а какие проигнорируете — до тех пор, пока такие решения принимаются осознанно.

Скажем, в примере приложения есть противоречие между паттерном "Хранилище", который стремится скрыть детали хранения данных, и реальностью работы с инфраструктурой Entity Framework Core.

За счет включения объекта `IQueryable<T>` в класс реализации хранилища ограничивается доля приложения, которой известно о запросах Entity Framework Core. Но ограничение сделано не полностью, потому что остаток приложения по-прежнему должен знать о свойстве первичного ключа, которое определено в листинге 5.3 и будет использоваться в последующих главах.

Для меня это разумный баланс между практичностью (объекты должны уникально идентифицироваться) и принципом (помещение деталей хранения данных в хранилище). Вы можете предпочесть вообще отказаться от применения хранилища или более строго придерживаться паттерна "Хранилище" (например, используя другую стратегию в отношении ключей, как описано в главе 19).

Альтернативный подход предполагает наличие класса реализации хранилища, который берет на себя ответственность иметь дело с индивидуальными особенностями объектов IQueryable<T> и предоставлять остальным частям приложения обычную коллекцию объектов в памяти, реализующую интерфейс IEnumerable<T>, которую можно перечислять, не беспокоясь о непредвиденных эффектах. В листинге 5.14 класс хранилища изменен, так что он больше не передает объекты DbSet<T>, возвращаемые свойством Products класса контекста.

Листинг 5.14. Принудительная оценка запроса в файле DataRepository.cs из папки Models

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products.ToArray();

    public void AddProduct(Product product) {
      this.context.Products.Add(product);
      this.context.SaveChanges();
    }
  }
}
```

Методы ToArray() и ToList() из LINQ запускают выполнение запроса и выпускают массив или список, который содержит результаты. Они представляют собой обычную коллекцию объектов в памяти, которая реализует только интерфейс IEnumerable<T>, что означает необходимость изменения модели представления в представлении Index, применяемом контроллером Home (листинг 5.15). Это также означает, что можно безопасно возвратиться к выполнению множества операций LINQ в представлении, не заботясь о том, как были получены данные.

На заметку! Одно из следствий такого подхода заключается в том, что хранилище должно быть в состоянии предоставлять оставшимся частям приложения данные, которые им требуются, в итоге приводя к объединению сложных запросов в классе хранилища. Я отдаю предпочтение такому подходу, поскольку он облегчает просмотр и управление всеми запросами, с которыми придется иметь дело инфраструктуре Entity Framework Core. Но это мое персональное предпочтение, а вы вольны выбирать тот подход, который лучше всего подходит именно вам.

Листинг 5.15. Изменение модели представления в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <div class="row">
```

```
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  <form asp-action="AddProduct" method="post">
    <div class="row">
      <div class="col"><input name="Name" class="form-control" /></div>
      <div class="col"><input name="Category" class="form-control" /></div>
      <div class="col">
        <input name="PurchasePrice" class="form-control" />
      </div>
      <div class="col">
        <input name="RetailPrice" class="form-control" />
      </div>
      <div class="col">
        <button type="submit" class="btn btn-primary">Add</button>
      </div>
    </div>
  </form>
  <div>
    @if (Model.Count() == 0) {
      <div class="row">
        <div class="col text-center p-2">No Data</div>
      </div>
    } else {
      @foreach (Product p in Model) {
        <div class="row p-2">
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col text-right">@p.PurchasePrice</div>
          <div class="col text-right">@p.RetailPrice</div>
          <div class="col"></div>
        </div>
      }
    }
  </div>
</div>
```

Перезапустите приложение, используя команду `dotnet run`, и перейдите по ссылке `http://localhost:5000`; вы увидите знакомый вывод, который был показан на рисунках ранее. Пользовательский интерфейс в текущем разделе не изменялся, но если вы просмотрите журнальные сообщения, сгенерированные приложением, то заметите, что в БД был отправлен только один запрос, хотя объект модели представления перечислялся дважды:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
...
```

Распространенные проблемы и их решения

Применение инфраструктуры Entity Framework Core для сохранения и извлечения данных прямолинейно после того, как основные средства на месте, но есть несколько ловушек, которых следует избегать. Далее будут описаны проблемы, с которыми вы вероятнее всего столкнетесь, и приведены объяснения, как их решить.

Проблемы, связанные с созданием или доступом к базе данных

Наиболее фундаментальные проблемы возникают при попытке создания БД или доступа к ней из приложения. Главным образом такие проблемы связаны с неправильной конфигурацией, как объясняется в последующих разделах.

Ошибка "No executable found matching command dotnet-ef" ("Не найден исполняемый файл, соответствующий команде dotnet-ef")

Команды `dotnet ef` используются для создания и управления миграциями, но по умолчанию они не включены и опираются на пакет, добавляемый в приложение. Если при попытке запуска любой команды `dotnet ef` вы получаете ошибку типа "Не найден исполняемый файл", тогда откройте файл `.csproj` и убедитесь в наличии ссылки `DotNetCliToolReference` для пакета `Microsoft.EntityFrameworkCore.Tools.DotNet`, как было показано в листинге 5.1.

Если вы добавили упомянутый пакет, то удостоверьтесь, что вводите команды в папке проекта, которая должна содержать файлы `.csproj` и `Startup.cs`. В случае применения `dotnet ef` в любой другой папке исполняющей среде .NET Core не удается найти используемые команды.

Ошибка "Build failed" ("Сборка потерпела неудачу")

При вводе команды `dotnet ef` проект автоматически компилируется, а ошибка "Сборка потерпела неудачу" возникает, если в коде обнаруживается любая проблема, хотя никаких деталей о причине проблемы не предоставляется.

Если вы хотите выяснить, что препятствует компиляции проекта, тогда введите в папке проекта команду `dotnet build`. Затем можете решить проблему и запустить команду `dotnet ef` снова.

На заметку! Ошибка "Сборка потерпела неудачу" также может быть вызвана вводом команд `dotnet ef` после запуска приложения с применением `dotnet run` в другом окне командной строки или PowerShell. Процесс сборки пытается переписать файлы, которые удерживаются открытыми выполняющимся приложением, что и становится причиной отказа. Остановите приложение и команда `dotnet ef` должна пройти успешно.

Ошибка "The entity type requires a primary key to be defined" ("Сущностный тип требует определения первичного ключа")

Такая ошибка возникает при попытке создания миграции вероятнее всего из-за того, что вы не выбрали первичный ключ. Для простых приложений наилучший подход был продемонстрирован в листинге 5.3. Для сложных приложений более развитые средства работы с ключами рассматриваются в главе 19.

Исключение "There is already an object named <Имя> in the database" ("В базе данных уже существует объект по имени <Имя>")

Это исключение генерируется в случае применения миграции, пытающейся создать таблицу БД, которая уже существует. Подобное обычно происходит, когда вы

удаляете миграцию из проекта, повторно создаете ее и затем пытаетесь снова применить к БД. В БД уже содержатся таблицы, созданные миграцией, что и препятствует успешному завершению.

Проблема чаще всего возникает на этапе разработки; простейшее решение предусматривает удаление и воссоздание БД за счет выполнения команд из листинга 5.16 в папке проекта. Команды удаляют БД вместе с содержащимися в ней данными и потому не должны использоваться в производственных системах.

Листинг 5.16. Переустановка БД

```
dotnet ef database drop --force
dotnet ef database update
```

Исключение "A Network-Related or Instance-Specific Error Occurred" ("Возникла ошибка, связанная с сетью или экземпляром")

Такое исключение говорит о том, что инфраструктура Entity Framework Core не смогла связаться с сервером баз данных. Самой распространенной причиной исключения является ошибка внутри строки подключения в файле `appsettings.json`. Если при разработке вы применяете LocalDB, тогда удостоверьтесь, что установили свойство конфигурации `Server` в строку `(localdb)//MSSQLLoclDb`, которая содержит два символа / и имя `MS_SQL_Local_Db`, но без символов подчеркивания. Если вы используете полный продукт SQL Server (или другой сервер баз данных), то проверьте правильность имени хоста и номера порта TCP, удостоверьтесь, что имя хоста преобразуется в корректный IP-адрес, и протестируйте сеть, убедившись в достижимости сервера.

Исключение "Cannot Open Database Requested By The Login" ("Не удается открыть базу данных, запрошенную входом")

Если вы получаете такое исключение, то инфраструктура Entity Framework Core в состоянии взаимодействовать с сервером баз данных, но запрашивает доступ к несуществующей БД. Первым делом следует удостовериться в том, что внутри строки подключения в файле `appsettings.json` указано правильное имя БД. Для LocalDB (и полных продуктов SQL Server) это означает корректную установку свойства `Database`, как показано в листинге 5.6. В случае применения другого сервера просмотрите в документации, как должно указываться имя БД.

Совет. Выяснить, что должны содержать строки подключения, может быть нелегко, особенно когда вы переходите на другой сервер баз данных или пакет поставщика. На веб-сайте `https://www.connectionstrings.com` предлагается удобный справочник по широкому диапазону серверов баз данных и вариантам подключения.

Если имя БД указано корректно, тогда вероятно вы создали миграцию, но не применили ее, а это значит, что у сервера баз данных не было затребовано создание БД, доступ к которой запрашивает инфраструктура Entity Framework Core. Чтобы применить миграцию, введите в папке проекта команду `dotnet ef database update`.

Проблемы, связанные с запрашиванием данных

Самой крупной проблемой при запрашивании данных являются дублированные запросы к серверу баз данных, как было описано в разделе "Избегание ловушек, связанных с запросами" ранее в главе. Но это не единственная проблема, которая может возникать, чему посвящены последующие разделы.

Исключение "Property Could Not Be Mapped" ("Свойство не может быт сопоставлено")

Такое исключение сгенерируется, если вы добавили свойство в класс модели данных, но не создали и не применили миграцию, чтобы обновить БД. В главе 13 подробно рассматривается, как использовать миграции для поддержания модели данных и БД в синхронизированном состоянии.

Исключение "Invalid Object Name" ("Недопустимое имя объекта")

Возникновение этого исключения обычно означает, что инфраструктура Entity Framework Core пыталась запросить данные из таблицы, которая в БД не существует. Оно представляет собой разновидность проблемы, описанной в предыдущем разделе, и в большинстве случаев говорит о том, что БД не была обновлена для отражения изменений, внесенных в модель данных приложения. Детальные сведения о работе миграций и управлении ими ищите в главе 13.

Исключение "There is Already an Open DataReader" ("Существует открытое средство чтения данных")

Такое исключение генерируется при попытке запуска запроса до того, как прочитаны все результаты из предыдущего запроса. Если вы работаете с SQL Server, то можете разрешить в строке подключения множественный активный набор данных (multiple active result set — MARS), как показано в листинге 5.6. Для других серверов баз данных вы можете применить метод `ToArray()` или `ToList()`, чтобы обеспечить чтение всех данных одним запросом перед началом следующего запроса.

Исключение "Cannot Consume Scoped Service from Singleton" ("Не удается потребить ограниченную службу из одиночки")

Метод `AddDbContext()` в классе `Startup` использует метод `AddedScoped()`, чтобы настроить средство внедрения зависимостей для класса контекста. Это означает, что для конфигурирования любой службы, которая зависит от класса контекста, вроде классов реализации хранилища, должен применяться метод `AddTransient()` или `AddScoped()`, как показано в листинге 5.7. Если для регистрации своих служб вы используете метод `AddSingleton()`, то при попытке распознавания зависимостей инфраструктурой ASP.NET Core возникнет исключение.

Проблема устаревшего контекста данных

В приложении ASP.NET Core MVC инфраструктура Entity Framework Core ожидает, что для каждого HTTP-запроса создается новый объект контекста. Тем не менее, распространенная проблема возникает, когда объекты контекста удерживают и пытаются их применять для последующих запросов.

Проблема такого подхода в том, что каждый объект контекста отслеживает созданные им объекты для использования кеша и обнаружения изменений.

Удержание объектов контекста и их повторное применение может привести к непредсказуемым результатам из-за устаревания или неполноты данных. Хотя вы можете испытывать антипатию к созданию объектов для одиночного запроса, это шаблон, используемый в остальных местах приложения (инфраструктура MVC создает новые объекты контроллера и представлений для каждого HTTP-запроса), и именно такого применения объектов контекста ожидает инфраструктура Entity Framework Core.

Проблемы, связанные с сохранением данных

В целом, чтобы дать возможность инфраструктуре Entity Framework Core сохранять экземпляры классов в модели данных MVC, требуется лишь несколько изменений. Однако могут возникать проблемы, которые описаны в последующих разделах.

Объекты не сохраняются

Если приложение выглядит работоспособным, но объекты не сохраняются в БД, тогда первым делом нужно проверить, что вы не забыли вызвать метод `SaveChanges()` в классе реализации хранилища. Инфраструктура Entity Framework Core будет обновлять БД только после вызова метода `SaveChanges()` и молча отбросит изменения, если забыть его вызвать.

Не сохраняются значения некоторых свойств

Если в БД сохраняются лишь некоторые значения данных, ассоциированные с объектом, то удостоверьтесь в том, что используются только свойства и все они имеют конструкции `set` и `get`. Инфраструктура Entity Framework Core по умолчанию будет сохранять только значения свойств, игнорируя любые методы или поля. Если ограничения вашего приложения препятствуют применению только свойств в классах модели данных, тогда ознакомьтесь с расширенными средствами Entity Framework Core в главе 20, чтобы изменить способ, которым используются классы модели данных.

Исключение "Cannot Insert Explicit Value for Identity Column" ("Не удается вставить явное значение для столбца идентичности")

Если вы выбрали первичный ключ, как показано в листинге 5.3, то инфраструктура Entity Framework Core сконфигурирует БД так, что сервер баз данных будет нести ответственность за генерацию значений, которые позволят объектам идентифицироваться уникальным образом. В итоге множество приложений — или экземпляров одного приложения — могут разделять ту же самую БД без координации во избежание дублированных значений. Кроме того, при попытке сохранения нового объекта со значением ключа, которое не является стандартным для типа ключа, будет сгенерировано исключение. Для класса `Product` первичный ключ имеет тип `long`, поэтому новые объекты могут сохраняться только с нулевым значением `Id`, принятым для типа `long` по умолчанию. Самая распространенная причина возникновения такого исключения связана с добавлением в представление, применяемое для создания новых объектов, элемента `input`, который позволяет пользователю вводить значение, впоследствии используемое связывателем моделей MVC и передаваемое в БД через Entity Framework Core.

Совет. Если вы не хотите, чтобы сервер баз данных генерировал идентифицирующие значения, тогда ознакомьтесь с расширенными вариантами для первичных ключей в главе 19.

Резюме

В главе была добавлена поддержка для сохранения данных в БД и выдачи запросов к ней. Приводились объяснения процесса перехода к постоянному хранилищу данных и демонстрации, каким образом запросы, инициируемые приложением, должны адаптироваться к эффективной работе с инфраструктурой Entity Framework Core. Были также описаны наиболее распространенные проблемы, с которыми вы вероятнее всего столкнетесь при внедрении Entity Framework Core в существующее приложение, и предложены их решения. В следующей главе к приложению `SportsStore` будут добавлены средства для модификации и удаления данных в БД.

ГЛАВА 6

SportsStore: модификация и удаление данных

В текущий момент приложение SportsStore способно сохранять объекты Product в БД и выполнять запросы для их чтения обратно в память. Большинству приложений также требуется возможность вносить изменения в данные после того, как они были сохранены, включая полное удаление объектов. В этой главе будет добавлена поддержка для обновления и удаления объектов Product. Вдобавок обсуждаются проблемы, с которыми вполне вероятно вы столкнетесь при добавлении таких средств в собственные проекты, и объясняется, как их решить.

Подготовительные шаги

Мы продолжим использовать проект SportsStore, созданный в главе 4, к которому в главе 5 была добавлена инфраструктура Entity Framework Core. Находясь в папке проекта SportsStore, выполните команды, показанные в листинге 6.1, чтобы удалить и заново создать БД, что поможет гарантировать получение ожидаемых результатов при проработке примеров.

Совет. Проект SportsStore и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Листинг 6.1. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением dotnet run и перейдите по ссылке http://localhost:5000; вы увидите содержимое, представленное на рис. 6.1.

Заполните HTML-форму, используя значения данных из табл. 6.1, что обеспечит данные для примеров, рассматриваемых в главе.

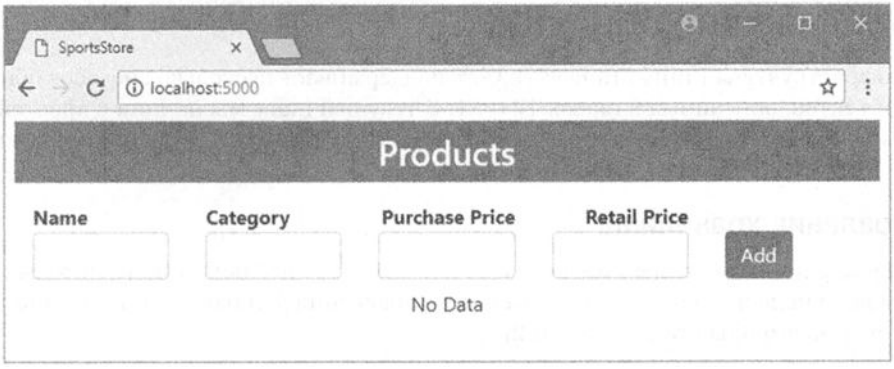

Рис. 6.1. Выполнение примера приложения

Таблица 6.1. Значения для создания тестовых объектов `Product`

Name **(Наименование)**	Category **(Категория)**	Purchase Price **(Покупная цена)**	Retail Price **(Розничная цена)**
`Kayak` (Каяк)	`Watersports` (Водный спорт)	200	275
`Lifejacket` (Спасательный жилет)	`Watersports` (Водный спорт)	30	48.95
`Soccer Ball` (Футбольный мяч)	`Soccer` (Футбол)	17	19.50

После ввода деталей всех трех товаров вы должны получить результаты, приведенные на рис. 6.2.

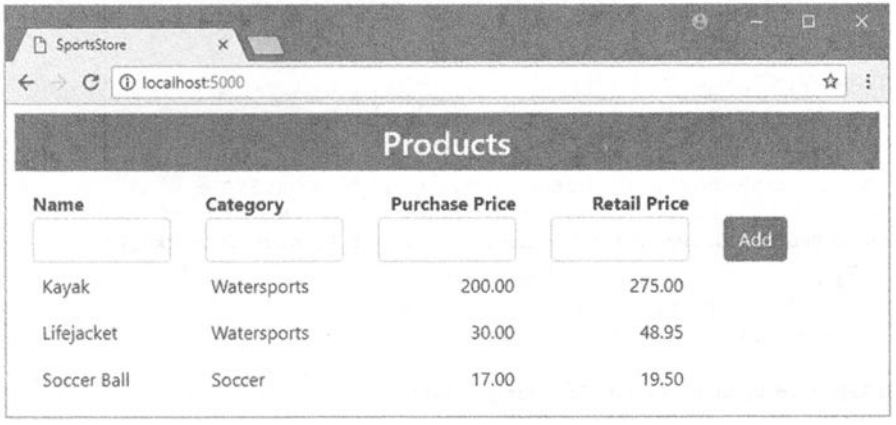

Рис. 6.2. Добавление тестовых данных

Модификация объектов

Инфраструктура Entity Framework Core поддерживает несколько способов обновления объектов, описанных в главах 12 и 21. В текущей главе мы начнем с простейшего приема, при котором объект, созданный связывателем моделей MVC, применяется для полной замены объекта, хранящегося в БД.

Обновление хранилища

Первым делом измените интерфейс `IRepository`, добавив к нему методы, которые можно использовать в остальном коде приложения для извлечения и обновления существующего объекта (листинг 6.2).

Листинг 6.2. Добавление методов в файле `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public interface IRepository {
    IEnumerable<Product> Products { get; }

    Product GetProduct(long key);

    void AddProduct(Product product);

    void UpdateProduct(Product product);
  }
}
```

Метод `GetProduct()` предоставит одиночный объект `Product` по значению его первичного ключа. Метод `UpdateProduct()` получает объект `Product` и какого-либо результата не возвращает. В листинге 6.3 новые методы добавляются и в класс реализации хранилища.

Листинг 6.3. Добавление методов в файле `DataRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;
    public IEnumerable<Product> Products => context.Products.ToArray();
    public Product GetProduct(long key) => context.Products.Find(key);

    public void AddProduct(Product product) {
      context.Products.Add(product);
      context.SaveChanges();
    }
    public void UpdateProduct(Product product) {
      context.Products.Update(product);
      context.SaveChanges();
    }
  }
}
```

Объект DbSet<Product>, возвращаемый свойством Products контекста БД, предоставляет функциональное средство, которое необходимо для реализации новых методов. Метод Find() принимает значение первичного ключа и запрашивает у БД соответствующий ему объект. Метод Update() принимает объект Product и применяет его для обновления БД, заменяя объект в БД с тем же самым первичным ключом. Как и в случае всех операций, которые изменяют БД, после вызова Update() понадобится вызвать метод SaveChanges().

Совет. Помнить о необходимости вызова метода SaveChanges() может показаться трудновыполнимой задачей, но это быстро войдет в привычку. Такой подход означает, что вы можете подготовить множество изменений, вызывая методы объекта контекста, и затем одновременно отправить их БД с помощью единственного вызова SaveChanges(). В главе 24 будут приведены подробные объяснения, как все работает.

Обновление контроллера и создание представления

Следующий шаг заключается в обновлении контроллера Home, чтобы добавить методы действий, которые позволят пользователю выбрать объект Product для редактирования и отправить изменения приложению (листинг 6.4). Кроме того, закомментирована строка, очищающая консоль, чтобы стало легче увидеть SQL-запросы, которые Entity Framework Core выполняет для новых методов.

Листинг 6.4. Добавление действий в файле HomeController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) => repository = repo;

    public IActionResult Index() {
      // System.Console.Clear();
      return View(repository.Products);
    }
    [HttpPost]
    public IActionResult AddProduct(Product product) {
      repository.AddProduct(product);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult UpdateProduct(long key) {
      return View(repository.GetProduct(key));
    }
    [HttpPost]
    public IActionResult UpdateProduct(Product product) {
      repository.UpdateProduct(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Вы можете заметить, как методы действий сопоставляются со средствами, предоставляемыми хранилищем, посредством класса контекста БД. Чтобы снабдить контроллер представлением для новых действий, добавьте в папку `Views/Home` файл по имени `UpdateProduct.cshtml` с содержимым из листинга 6.5.

Листинг 6.5. Содержимое файла `UpdateProduct.cshtml` из папки `Views/Home`

```
@model Product

<h3 class="p-2 bg-primary text-white text-center">Update Product</h3>

<form asp-action="UpdateProduct" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" readonly />
  </div>

  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </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="PurchasePrice"></label>
    <input asp-for="PurchasePrice" class="form-control" />
  </div>

  <div class="form-group">
    <label asp-for="RetailPrice"></label>
    <input asp-for="RetailPrice" 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>
```

Представление `UpdateProduct.cshtml` снабжает пользователя HTML-формой, которую можно использовать для изменения свойств объекта `Product` за исключением свойства `Id`, применяемого в качестве первичного ключа. Первичные ключи нелегко изменить после того, как они были назначены, и если требуется другое значение ключа, то проще удалить объект и создать новый. По этой причине к элементу `input` добавлен атрибут `readonly`, который позволяет видеть значение свойства `Id`, но не изменять его.

Для интеграции средства обновления с остальным кодом приложения добавьте элемент `button` к каждому объекту `Product`, отображаемому представлением `Index` (листинг 6.6). Также добавьте в сетку столбец, который будет отображать свойство `Id`.

Листинг 6.6. Интеграция обновлений в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <div class="row">
   <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  <form asp-action="AddProduct" method="post">
   <div class="row p-2">
    <div class="col-1"></div>
     <div class="col"><input name="Name" class="form-control" /></div>
     <div class="col"><input name="Category" class="form-control" /></div>
     <div class="col">
       <input name="PurchasePrice" class="form-control" />
     </div>
     <div class="col">
       <input name="RetailPrice" class="form-control" />
     </div>
     <div class="col">
       <button type="submit" class="btn btn-primary">Add</button>
     </div>
   </div>
  </form>
  <div>
    @if (Model.Count() == 0) {
      <div class="row">
        <div class="col text-center p-2">No Data</div>
      </div>
    } else {
      @foreach (Product p in Model) {
        <div class="row p-2">
         <div class="col-1">@p.Id</div>
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col text-right">@p.PurchasePrice</div>
          <div class="col text-right">@p.RetailPrice</div>
         <div class="col">
           <a asp-action="UpdateProduct" asp-route-key="@p.Id"
              class="btn btn-outline-primary">
             Edit
           </a>
         </div>
        </div>
      }
    }
  </div>
</div>
```

Запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`; вы увидите новые элементы, которые отображают первичный ключ и предоставляют кнопки Edit (Редактировать) для всех товаров (рис. 6.3).

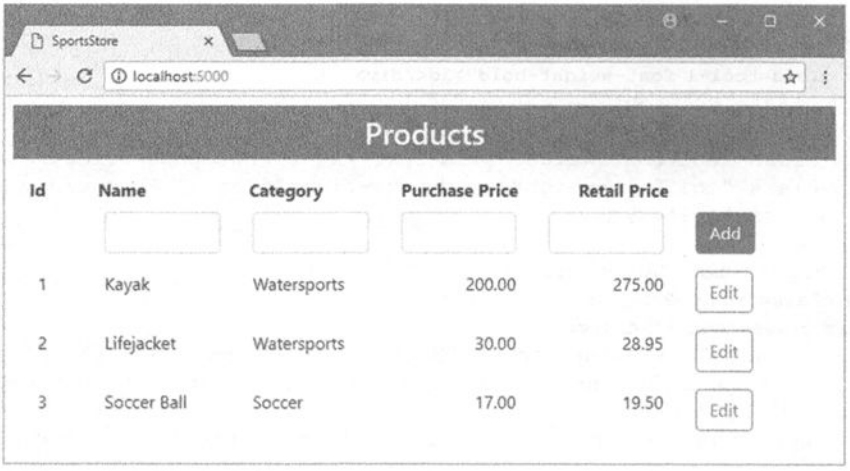

Рис. 6.3. Добавление элементов к представлению `Index`

Щелкните на кнопке Edit для товара `Soccer Ball`, измените значение в поле Purchase Price (Покупная цена) на `16.50` и щелкните на кнопке Save (Сохранить). Браузер отправит данные формы методу действия `UpdateProduct()` контроллера `Home`, который получит объект `Product`, созданный связывателем моделей MVC. Объект `Product` будет передан методу `Update()` класса контекста БД, и когда вызывается метод `SaveChanges()`, значения данных формы сохраняются в БД (рис. 6.4).

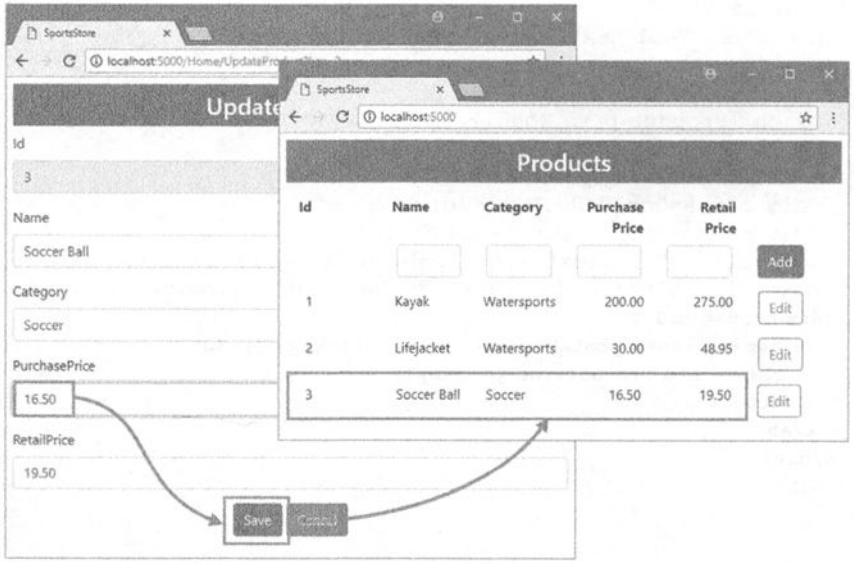

Рис. 6.4. Обновление объекта

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то сможете увидеть, как выполняемые действия превращаются в SQL-команды, отправляемые серверу баз данных. Когда вы щелкаете на кнопке Edit, инфраструктура Entity Framework Core запрашивает у БД детали объекта `Soccer Ball` с помощью следующей команды:

```
...
SELECT TOP(1) [e].[Id], [e].[Category], [e].[Name], [e].[PurchasePrice],
  [e].[RetailPrice]
FROM [Products] AS [e]
WHERE [e].[Id] = @__get_Item_0
...
```

Метод `Find()`, применяемый в листинге 6.6, транслируется в команду `SELECT` для одиночного объекта, который указан с использованием ключевого слова `TOP`. После щелчка на кнопке Save инфраструктура Entity Framework Core обновляет БД с помощью такой команды:

```
...
UPDATE [Products] SET [Category] = @p0, [Name] = @p1,
  [PurchasePrice] = @p2, [RetailPrice] = @p3
WHERE [Id] = @p4;
...
```

Метод `Update()` транслируется в SQL-команду `UPDATE`, которая сохраняет значения данных формы, полученные из HTTP-запроса.

Обновление только измененных свойств

Основные строительные блоки для выполнения обновлений на месте, но результат неэффективен, поскольку инфраструктура Entity Framework Core не располагает исходными данными для оценки, что конкретно изменилось, и не имеет другого выбора, кроме сохранения всех свойств. Чтобы взглянуть на проблему, щелкните на кнопке Edit для одного из товаров и затем щелкните на кнопке Save, не внося каких-либо изменений. Хотя новые значения данных не вводились, журнальные сообщения, сгенерированные приложением, говорят о том, что выпущенная инфраструктурой Entity Framework Core команда `UPDATE` посылает значения для всех свойств, которые определены в классе `Product`:

```
...
UPDATE [Products] SET [Category] = @p0, [Name] = @p1, [PurchasePrice] = @p2,
  [RetailPrice] = @p3
WHERE [Id] = @p4;
...
```

Инфраструктура Entity Framework Core содержит средство обнаружения изменений, которое способно выявить, какие свойства были изменены. Для такого простого класса модели данных, каковым является `Product`, польза от этого средства невелика, но в случае более сложных моделей обнаружение изменений может быть важным.

Средство обнаружения изменений требует исходных данных, с которыми можно было бы сравнить данные, полученные от пользователя. Есть разные способы предоставления исходных данных (они описаны в главе 12), но здесь будет применен простейший подход, который предусматривает запрашивание у БД существующих данных. В листинге 6.7 класс реализации хранилища модифицирован так, чтобы запрашивать у БД сохраненный объект `Product` и использовать его для обновления только измененных свойств.

Совет. Издержки на запрос и выгода от избегания нежелательных обновлений должны быть сбалансированными. Тем не менее, такой подход прост и надежен, а также хорошо работает с функциональными средствами Entity Framework Core, которые предназначены для того, чтобы не допустить обновление одних и тех же данных двумя разными пользователями (они описаны в главе 20).

Листинг 6.7. Избегание нежелательных обновлений в файле `DataRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products.ToArray();

    public Product GetProduct(long key) => context.Products.Find(key);

    public void AddProduct(Product product) {
      context.Products.Add(product);
      context.SaveChanges();
    }

    public void UpdateProduct(Product product) {
      Product p = GetProduct(product.Id);
      p.Name = product.Name;
      p.Category = product.Category;
      p.PurchasePrice = product.PurchasePrice;
      p.RetailPrice = product.RetailPrice;
      // context.Products.Update(product);
      context.SaveChanges();
    }
  }
}
```

Код объединяет в приложении две разных функции. Инфраструктура Entity Framework Core отслеживает изменения в объектах, которые создает из данных запросов, в то время как связыватель моделей MVC создает объекты из HTTP-данных. Упомянутые два источника объектов не объединены, и если не соблюдать осторожность при удержании их раздельными, то возникнут проблемы. Самый безопасный способ задействовать в своих интересах отслеживание изменений — запросить БД и затем скопировать значения из HTTP-данных, как было сделано в листинге 6.7. Когда вызывается метод `SaveChanges()`, инфраструктура Entity Framework Core выяснит, значения каких свойств изменились, и обновит в БД только эти свойства.

Совет. Обратите внимание, что вызов метода `Update()` закомментирован, т.к. он не нужен, когда запрос предоставляет исходные данные.

Чтобы посмотреть, как работает приложение, запустите его с применением команды `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit для товара `Kayak`. Измените значение Retail Price (Розничная цена) на `300` и щелкните на кнопке Save. В журнальных сообщениях, сгенерированных приложением, вы увидите, что команда `UPDATE`, которую инфраструктура Entity Framework Core отправила БД, модифицирует только измененное свойство:

```
...
UPDATE [Products] SET [RetailPrice] = @p0
WHERE [Id] = @p1;
...
```

Выполнение массовых обновлений

Массовые обновления часто требуются в приложениях, где предусмотрены отдельные роли администраторов, которым необходимо вносить изменения во множество объектов в единственной операции. Точная природа обновлений будет варьироваться, но распространенные причины для массовых обновлений включают исправление ошибок в записях данных или переназначение объектов в новые категории, что может оказаться затратным по времени процессом в случае выполнения на индивидуальных объектах. С использованием инфраструктуры Entity Framework Core делать массовые обновления легко, но требуются небольшие усилия, чтобы обеспечить их гладкую работу с частью ASP.NET Core MVC приложения.

Изменение представлений и контроллера

Для добавления поддержки массовых обновлений измените представление `Index`, включив в него кнопку Edit All (Редактировать все), которая нацелена на действие `UpdateAll`. Также добавьте свойство типа `ViewBag` по имени `UpdateAll`; если его значением является `true`, тогда будет отображаться частичное представление `InlineEditor.cshtml` (листинг 6.8).

Листинг 6.8. Поддержка массовых обновлений в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  @if (ViewBag.UpdateAll != true) {
    <div class="row">
      <div class="col-1 font-weight-bold">Id</div>
      <div class="col font-weight-bold">Name</div>
      <div class="col font-weight-bold">Category</div>
      <div class="col font-weight-bold text-right">Purchase Price</div>
      <div class="col font-weight-bold text-right">Retail Price</div>
      <div class="col"></div>
    </div>
    <form asp-action="AddProduct" method="post">
      <div class="row p-2">
        <div class="col-1"></div>
        <div class="col"><input name="Name" class="form-control" /></div>
        <div class="col"><input name="Category" class="form-control" /></div>
```

```
        <div class="col">
          <input name="PurchasePrice" class="form-control" />
        </div>
        <div class="col">
          <input name="RetailPrice" class="form-control" />
        </div>
        <div class="col">
          <button type="submit" class="btn btn-primary">Add</button>
        </div>
      </div>
    </form>
    <div>
      @if (Model.Count() == 0) {
        <div class="row">
          <div class="col text-center p-2">No Data</div>
        </div>
      } else {
        @foreach (Product p in Model) {
          <div class="row p-2">
            <div class="col-1">@p.Id</div>
            <div class="col">@p.Name</div>
            <div class="col">@p.Category</div>
            <div class="col text-right">@p.PurchasePrice</div>
            <div class="col text-right">@p.RetailPrice</div>
            <div class="col">
              <a asp-action="UpdateProduct" asp-route-key="@p.Id"
                 class="btn btn-outline-primary">
                Edit
              </a>
            </div>
          </div>
        }
      }
    </div>
    <div class="text-center">
      <a asp-action="UpdateAll" class="btn btn-primary">Edit All</a>
    </div>
  } else {
    @Html.Partial("InlineEditor", Model)
  }
</div>
```

Создайте частичное представление, добавив в папку `Views/Home` файл по имени `InlineEditor.cshtml` с содержимым из листинга 6.9.

Листинг 6.9. Содержимое файла `InlineEditor.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<div class="row">
  <div class="col-1 font-weight-bold">Id</div>
  <div class="col font-weight-bold">Name</div>
  <div class="col font-weight-bold">Category</div>
  <div class="col font-weight-bold">Purchase Price</div>
```

```
  <div class="col font-weight-bold">Retail Price</div>
</div>
@{ int i = 0; }
<form asp-action="UpdateAll" method="post">
  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col-1">
        @p.Id
        <input type="hidden" name="Products[@i].Id" value="@p.Id" />
      </div>
      <div class="col">
        <input class="form-control" name="Products[@i].Name"
              value="@p.Name" />
      </div>
      <div class="col">
        <input class="form-control" name="Products[@i].Category"
              value="@p.Category" />
      </div>
      <div class="col text-right">
        <input class="form-control" name="Products[@i].PurchasePrice"
              value="@p.PurchasePrice" />
      </div>
      <div class="col text-right">
        <input class="form-control" name="Products[@i].RetailPrice"
              value="@p.RetailPrice" />
      </div>
    </div>
    i++;
  }
  <div class="text-center m-2">
    <button type="submit" class="btn btn-primary">Save All</button>
    <a asp-action="Index" class="btn btn-outline-primary">Cancel</a>
  </div>
</form>
```

Частичное представление создает набор элементов формы, имена которых следуют соглашению, принятому в MVC для коллекции объектов, а потому свойству `Id` назначаются имена `Products[0].Id`, `Products[1].Id` и т.д. Установка имен для элементов `input` требует счетчика, что порождает неуклюжую смесь выражений Razor и C#.

Добавьте в контроллер `Home` методы действий, которые позволят пользователю начать процесс массового редактирования и отправить данные (листинг 6.10).

Листинг 6.10. Добавление методов действий в файле `HomeController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
```

```
    public HomeController(IRepository repo) => repository = repo;
    public IActionResult Index() {
      // System.Console.Clear();
      return View(repository.Products);
    }
    [HttpPost]
    public IActionResult AddProduct(Product product) {
      repository.AddProduct(product);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult UpdateProduct(long key) {
      return View(repository.GetProduct(key));
    }
    [HttpPost]
    public IActionResult UpdateProduct(Product product) {
      repository.UpdateProduct(product);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult UpdateAll() {
      ViewBag.UpdateAll = true;
      return View(nameof(Index), repository.Products);
    }
    [HttpPost]
    public IActionResult UpdateAll(Product[] products) {
      repository.UpdateAll(products);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Версия `POST` метода `UpdateAll()` принимает массив объектов `Product`, которые связыватель моделей MVC создает из данных формы, и передает его методу хранилища с тем же самым именем.

Изменение хранилища

Добавьте в интерфейс хранилища новый метод, который будет выполнять массовое обновление (листинг 6.11).

Листинг 6.11. Добавление метода в файле `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {
  public interface IRepository {
    IEnumerable<Product> Products { get; }
    Product GetProduct(long key);
    void AddProduct(Product product);
    void UpdateProduct(Product product);
    void UpdateAll(Product[] products);
  }
}
```

Наконец, добавьте в реализацию хранилища метод `UpdateAll()`, который будет обновлять БД с применением данных, полученных из HTTP-запроса (листинг 6.12).

Листинг 6.12. Выполнение массового редактирования в файле `DataRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products.ToArray();

    public Product GetProduct(long key) => context.Products.Find(key);

    public void AddProduct(Product product) {
      context.Products.Add(product);
      context.SaveChanges();
    }

    public void UpdateProduct(Product product) {
      Product p = GetProduct(product.Id);
      p.Name = product.Name;
      p.Category = product.Category;
      p.PurchasePrice = product.PurchasePrice;
      p.RetailPrice = product.RetailPrice;
      // context.Products.Update(product);
      context.SaveChanges();
    }

    public void UpdateAll(Product[] products) {
      context.Products.UpdateRange(products);
      context.SaveChanges();
    }
  }
}
```

Класс `DbSet<T>` предлагает методы для работы с индивидуальными объектами и с коллекциями объектов. В этом примере используется метод `UpdateRange()`, который является аналогом метода `Update()`, но работает с коллекциями. Когда вызывается метод `SaveChanges()`, инфраструктура Entity Framework Core отправляет последовательность SQL-команд `UPDATE` для обновления БД. Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit All, чтобы получить доступ к средству массового редактирования (рис. 6.5).

Использование средства обнаружения изменений для массовых обновлений

В листинге 6.12 не применяется средство обнаружения изменений Entity Framework Core, т.е. будут обновляться все свойства всех объектов `Product`. Чтобы обновлять только изменившиеся значения, модифицируйте метод `UpdateAll()` в классе хранилища, как показано в листинге 6.13.

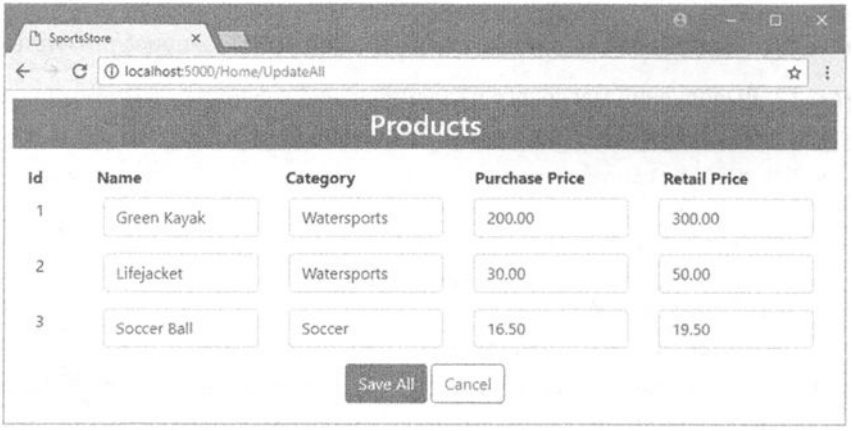

Рис. 6.5. Редактирование множества объектов

Листинг 6.13. Использование средства обнаружения изменений в файле DataRepository.cs из папки Models

```
...
public void UpdateAll(Product[] products) {
  // context.Products.UpdateRange(products);

  Dictionary<long, Product> data = products.ToDictionary(p => p.Id);
  IEnumerable<Product> baseline =
    context.Products.Where(p => data.Keys.Contains(p.Id));

  foreach(Product databaseProduct in baseline) {
    Product requestProduct = data[databaseProduct.Id];
    databaseProduct.Name = requestProduct.Name;
    databaseProduct.Category = requestProduct.Category;
    databaseProduct.PurchasePrice = requestProduct.PurchasePrice;
    databaseProduct.RetailPrice = requestProduct.RetailPrice;
  }
  context.SaveChanges();
}
...
```

Процесс выполнения обновления может выглядеть запутанным. Сначала создается словарь объектов `Product`, полученных от связывателя моделей MVC, с применением свойства `Id` для ключей. Коллекция ключей используется для запрашивания соответствующих объектов в БД:

```
...
IEnumerable<Product> baseline =
  context.Products.Where(p => data.Keys.Contains(p.Id));
...
```

Далее происходит перечисление объектов БД с копированием значений свойств из объектов HTTP-запроса. После вызова метода `SaveChanges()` инфраструктура Entity Framework Core выполняет обнаружение изменений и обновляет только те свойства, значения которых изменились. Запустите приложение, применив команду `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit All.

Измените значение в поле Name (Наименование) первого товара на `Green Kayak` (Зеленый каяк) и значение в поле Retail Price для товара `Lifejacket` на `50`. Щелкните на кнопке Save All (Сохранить все) и просмотрите журнальные сообщения, сгенерированные приложением. Для получения исходных данных для обнаружения изменений инфраструктура Entity Framework Core посылает БД следующий запрос:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[PurchasePrice],
[p].[RetailPrice]
FROM [Products] AS [p]
WHERE [p].[Id] IN (1, 2, 3)
...
```

Объекты, созданные из этих данных, используются для обнаружения изменений. Инфраструктура Entity Framework Core выясняет, какие свойства имеют новые значения, и отправляет БД две команды `UPDATE`:

```
...
UPDATE [Products] SET [Name] = @p0
WHERE [Id] = @p1;
...
UPDATE [Products] SET [RetailPrice] = @p2
WHERE [Id] = @p3;
...
```

Вы видите, что значение `Name` изменяется первой командой, а значение `RetailPrice` — второй командой, что соответствует изменениям, инициированным с применением части MVC приложения.

Удаление данных

Удаление объектов из БД — процесс простой, хотя с разрастанием модели данных он может усложниться, как объяснялось в главе 7. В листинге 6.14 к интерфейсу хранилища добавлен метод `Delete()`.

Листинг 6.14. Добавление метода в файле `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {
  public interface IRepository {
    IEnumerable<Product> Products { get; }
    Product GetProduct(long key);
    void AddProduct(Product product);
    void UpdateProduct(Product product);
    void UpdateAll(Product[] products);
    void Delete(Product product);
  }
}
```

В листинге 6.15 к классу реализации хранилища добавляется поддержка метода `Delete()`.

Листинг 6.15. Удаление объектов в файле DataRepository.cs из папки Models

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products.ToArray();

    public Product GetProduct(long key) => context.Products.Find(key);

    // ...для краткости остальные методы не показаны...

    public void Delete(Product product) {
      context.Products.Remove(product);
      context.SaveChanges();
    }
  }
}
```

Класс DbSet<T> имеет методы Remove() и RemoveRange(), предназначенные для удаления одного и нескольких объектов из БД. Как и в случае других операций, модифицирующих БД, данные не удаляются до тех пор, пока не будет вызван метод SaveChanges().

Добавьте в контроллер Home метод действия, который получает из HTTP-запроса детали подлежащего удалению объекта Product и передает их хранилищу (листинг 6.16).

Листинг 6.16. Добавление метода действия в файле HomeController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) => repository = repo;

    // ...для краткости остальные методы не показаны...

    [HttpPost]
    public IActionResult Delete(Product product) {
      repository.Delete(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы завершить функциональное средство, для каждого объекта Product, который отображается представлением Index, используемым контроллером Home, добавьте элемент form, как показано в листинге 6.17.

Совет. Форма содержит существующий элемент кнопки Edit, так что браузер будет отображать две кнопки рядом.

Листинг 6.17. Добавление элемента `form` в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  @if (ViewBag.UpdateAll != true) {
    <div class="row">
      <div class="col-1 font-weight-bold">Id</div>
      <div class="col font-weight-bold">Name</div>
      <div class="col font-weight-bold">Category</div>
      <div class="col font-weight-bold text-right">Purchase Price</div>
      <div class="col font-weight-bold text-right">Retail Price</div>
      <div class="col"></div>
    </div>
    <form asp-action="AddProduct" method="post">
      <div class="row p-2">
        <div class="col-1"></div>
        <div class="col"><input name="Name" class="form-control" /></div>
        <div class="col"><input name="Category" class="form-control" /></div>
        <div class="col">
          <input name="PurchasePrice" class="form-control" />
        </div>
        <div class="col">
          <input name="RetailPrice" class="form-control" />
        </div>
        <div class="col">
          <button type="submit" class="btn btn-primary">Add</button>
        </div>
      </div>
    </form>
    <div>
      @if (Model.Count() == 0) {
        <div class="row">
          <div class="col text-center p-2">No Data</div>
        </div>
      } else {
        @foreach (Product p in Model) {
          <div class="row p-2">
            <div class="col-1">@p.Id</div>
            <div class="col">@p.Name</div>
            <div class="col">@p.Category</div>
            <div class="col text-right">@p.PurchasePrice</div>
            <div class="col text-right">@p.RetailPrice</div>
            <div class="col">
            <form asp-action="Delete" method="post">
               <a asp-action="UpdateProduct" asp-route-key="@p.Id"
                  class="btn btn-outline-primary">
```

```
                Edit
              </a>
            <input type="hidden" name="Id" value="@p.Id" />
            <button type="submit" class="btn btn-outline-danger">
              Delete
            </button>
          </form>
          </div>
        </div>
      }
    }
  </div>
  <div class="text-center">
    <a asp-action="UpdateAll" class="btn btn-primary">Edit All</a>
  </div>
 } else {
  @Html.Partial("InlineEditor", Model)
 }
</div>
```

Обратите внимание, что форма содержит единственный элемент `input` для свойства `Id`. Это все, что инфраструктура Entity Framework Core применяет для удаления объекта из БД, хотя операция выполняется над полным объектом `Product`. Вместо отправки дополнительных данных, которые не планируется использовать, было послано только значение первичного ключа, которое связыватель моделей MVC применит для создания объекта `Product`, оставляя все другие свойства равными `null` или стандартному значению для типа.

На заметку! Мы имеем еще один пример решения проблемы, скольким деталям реализации разрешено просачиваться в остальные части приложения. Отправка для операции удаления одного лишь значения `Id` рациональна и проста, но полагается на знание того, как работает инфраструктура Entity Framework Core, и создает зависимость от способа хранения данных. В качестве альтернативы можно не опираться на поведение Entity Framework Core, но тогда придется отправлять значения для свойств, которые будут проигнорированы, и тем самым увеличить ширину полосы пропускания, требующуюся приложению. Одни проектные решения являются ясно очерченными, тогда как другие требуют сложного выбора из субоптимальных вариантов.

Чтобы протестировать функцию удаления, запустите приложение с помощью `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete (Удалить) для товара `Soccer Ball`. Объект `Product` будет удален из БД (рис. 6.6).

Распространенные проблемы и их решения

Средства Entity Framework Core для обновления и удаления данных достаточно прямолинейны, хотя могут возникать затруднения при их использовании с объектами, которые были созданы связывателем моделей MVC из HTTP-запросов. В последующих разделах описаны проблемы, с которыми вы вероятнее всего столкнетесь, и приведены объяснения, как их решить.

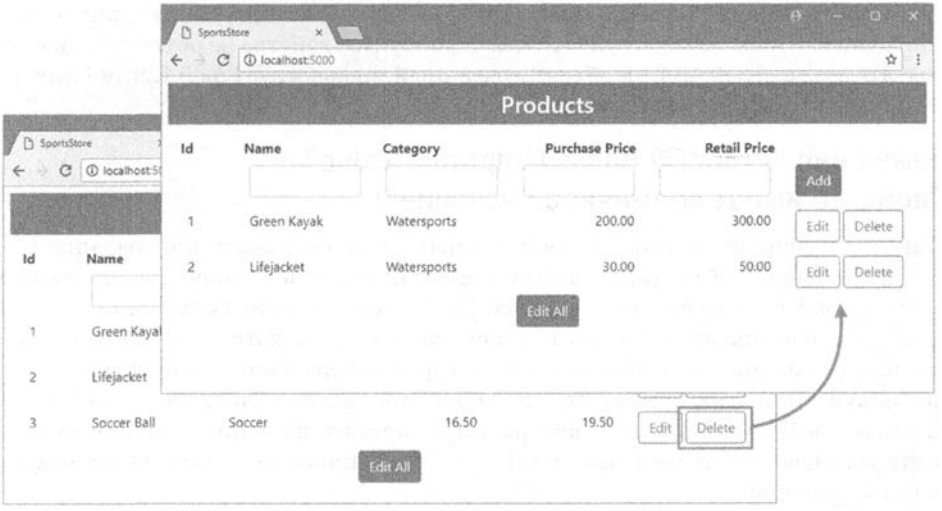

Рис. 6.6. Удаление объекта из БД

Объекты не обновляются или не удаляются

Если приложение выглядит работоспособным, но объекты не модифицируются, тогда первым делом нужно проверить, вызывается ли метод `SaveChanges()` в классе реализации хранилища. Инфраструктура Entity Framework Core будет обновлять БД только после вызова метода `SaveChanges()` и молча отбрасывать изменения, если вызов отсутствует.

Исключение "Reference Not Set to an Instance of an Object" ("Не установлена ссылка на экземпляр объекта")

Это исключение возникает из-за попытки обновить объект, свойство первичного ключа которого установлено в `null` или `0`. Самая распространенная причина такой проблемы объясняется тем, что вы забыли включить значение для свойства первичного ключа в HTML-форму, применяемую для обновления объекта. Несмотря на невозможность изменения значения первичного ключа, вы должны обеспечить его предоставление в виде части HTML-формы. Если вы не хотите, чтобы пользователь видел значение первичного ключа, тогда используйте скрытый элемент `input`.

Исключение "Instance of Entity Type Cannot be Tracked" ("Не удалось отследить экземпляр сущностного типа")

Это исключение возникает, когда производится вызов метода `Update()` инфраструктуры Entity Framework Core с применением объекта, который создан связывателем моделей MVC после запрашивания у БД того же самого объекта, используя Entity Framework Core. Класс контекста БД отслеживает создаваемые объекты, выполняя работу по обнаружению изменений, и инфраструктура Entity Framework Core не сможет справиться с ситуацией, когда вы вводите конфликтующий объект, который был создан инфраструктурой MVC.

Применять метод `Update()` можно только в случае, если исходные данные не запрашивались. Чтобы избежать проблемы, скопируйте свойства из объекта, созданного связывателем моделей MVC, в объект, созданный инфраструктурой Entity Framework Core, как было показано в листинге 6.7.

Исключение "Property Has a Temporary Value" ("Свойство имеет временное значение")

Такое исключение возникает, когда вы пытаетесь отправить приложению HTTP-запрос на удаление объекта, но забываете включить в него значение для свойства первичного ключа. Связыватель моделей MVC создаст объект со значением первичного ключа, являющимся стандартным для типа свойства, которое служит признаком временного значения, ожидающего замены сервером баз данных при сохранении нового объекта. Чтобы предотвратить генерацию этого исключения, обеспечьте наличие в HTML-форме элемента `input`, который предоставит значение первичного ключа. Можете установить тип элемента `input` в `hidden`, лишив пользователя возможности изменять значение.

Обновления в результате дают нулевые значения

Если в результате обновления числовое свойство получает нулевое значение, тогда вероятнее всего причина в том, что HTML-форма не включает значение для этого свойства или же связыватель моделей MVC не смог преобразовать введенное пользователем значение в тип данных свойства. Чтобы устранить первую проблему, удостоверьтесь в наличии значений для всех свойств, определенных классом модели данных. Чтобы исправить вторую проблему, используйте средства проверки достоверности MVC, обеспечивая частичные обновления, когда значения данных не могут быть обработаны.

Резюме

В главе к приложению `SportsStore` была добавлена поддержка для обновления и удаления объектов. Вы узнали, каким образом модифицировать индивидуальные объекты и выполнять массовые обновления, а также как снабжать инфраструктуру Entity Framework Core исходными данными для ее средства обнаружения изменений. Кроме того, было показано, как удалять данные, что делается просто в случае модели данных с единственным классом и усложняется с разрастанием модели данных. В следующей главе модель данных для приложения `SportsStore` будет расширена.

ГЛАВА 7

SportsStore: расширение модели данных

В этой главе модель данных для приложения SportsStore будет расширена за рамки единственного класса Product. Вы увидите, как нормализовать данные, заменяя строковое свойство отдельным классом, и узнаете, каким образом получать доступ к данным после их создания. Кроме того, добавляется поддержка для представления заказов покупателей, которая является важной частью любого Интернет-магазина.

Подготовительные шаги

Мы продолжим использовать проект SportsStore, который был создан в главе 4 и модифицирован в последующих главах. В рамках подготовки мы консолидируем процесс создания и редактирования объектов Product в единственном представлении. В листинге 7.1 мы объединяем методы действий контроллера Home, добавляющие или обновляющие объекты Product, и удаляем действия, которые выполняли массовые обновления.

Совет. Проект SportsStore и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Листинг 7.1. Консолидация действий в файле HomeController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;

    public HomeController(IRepository repo) => repository = repo;

    public IActionResult Index() {
      return View(repository.Products);
    }
```

```
    public IActionResult UpdateProduct(long key) {
      return View(key == 0 ? new Product() : repository.GetProduct(key));
    }
    [HttpPost]
    public IActionResult UpdateProduct(Product product) {
      if (product.Id == 0) {
        repository.AddProduct(product);
      } else {
        repository.UpdateProduct(product);
      }
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Product product) {
      repository.Delete(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

При выяснении, желает ли пользователь модифицировать существующий объект или же создать новый, объединенные действия опираются на стандартное значение для свойств `long`. В листинге 7.2 приведено обновленное представление `Index`, которое отражает изменения, внесенные в контроллер.

Листинг 7.2. Отражение изменений, внесенных в контроллер, в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col-1">@p.Id</div>
      <div class="col">@p.Name</div>
      <div class="col">@p.Category</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
      <div class="col">
        <form asp-action="Delete" method="post">
          <a asp-action="UpdateProduct" asp-route-key="@p.Id"
             class="btn btn-outline-primary">
            Edit
          </a>
```

```
        <input type="hidden" name="Id" value="@p.Id" />
        <button type="submit" class="btn btn-outline-danger">
          Delete
        </button>
      </form>
    </div>
  </div>
}
<div class="text-center p-2">
  <a asp-action="UpdateProduct" asp-route-key="0"
    class="btn btn-primary">Add</a>
</div>
</div>
```

Находясь в папке проекта `SportsStore`, выполните команды из листинга 7.3, чтобы удалить и повторно создать БД, что поможет гарантировать получение ожидаемых результатов при проработке примеров.

Листинг 7.3. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`; вы увидите содержимое, представленное на рис. 7.1. Добавлять какие-либо данные в БД сейчас не следует, потому что далее она будет обновлена посредством новой миграции, и добавление данных приведет к исключению.

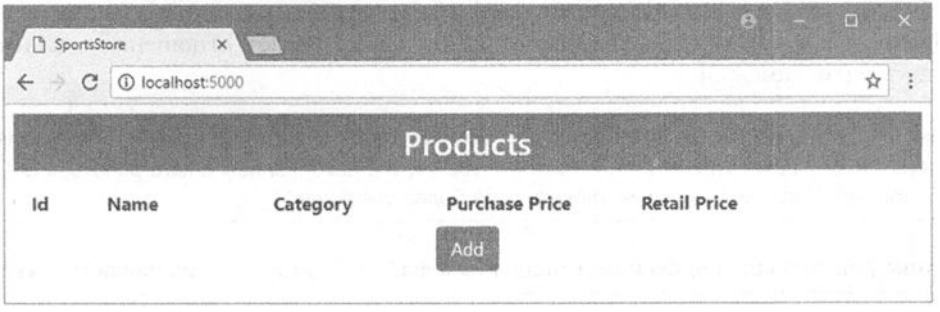

Рис. 7.1. Выполнение примера приложения

Добавление отношения в модель данных

В настоящий момент каждый объект `Product` создается со значением свойства `Category`, которое имеет тип `string`. В реальном проекте вопрос лишь в том, сколько времени пройдет до ситуации, когда из-за опечатки товар будет помещен в неподходящую категорию. Во избежание проблемы подобного рода данные приложения можно нормализовать с использованием отношений, в итоге сокращая дублирование и гарантируя согласованность, что и демонстрируется в последующих разделах.

Добавление класса модели данных

Отправной точкой станет создание нового класса модели данных. Добавьте в папку `Models` файл по имени `Category.cs` и определите в нем класс, как показано в листинге 7.4.

Листинг 7.4. Содержимое файла `Category.cs` из папки `Models`

```
namespace SportsStore.Models {
  public class Category {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
  }
}
```

Класс `Category` представляет категорию товаров. Свойство `Id` содержит первичный ключ, а значения для свойств `Name` и `Description` будут предоставлены пользователем при создании новой категории и ее сохранении в БД.

Создание отношения

На следующем шаге создается отношение между двумя классами модели данных, что делается путем добавления свойств к одному из классов. В любом отношении между данными один из классов известен как зависимая сущность и именно к нему добавляются свойства. Чтобы выяснить, какой класс является зависимой сущностью, задайте себе вопрос, объект какого из типов не может существовать без другого. В случае приложения `SportsStore` категория способна существовать, даже не имея в себе товаров, но каждый товар должен принадлежать какой-нибудь категории — и это значит, что в такой ситуации зависимой сущностью оказывается класс `Product`. Добавьте в класс `Product` два свойства, которые создают отношение с классом `Category` (листинг 7.5).

Совет. Не переживайте, если в настоящий момент выбор зависимой сущности не вполне понятен. Мы возвратимся к этой теме в главе 14, и с накоплением опыта работы с Entity Framework Core вы будете чувствовать себя более уверенно.

Листинг 7.5. Добавление свойств отношения в файле `Product.cs` из папки `Models`

```
namespace SportsStore.Models {
  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    // public string Category { get; set; }
    public decimal PurchasePrice { get; set; }
    public decimal RetailPrice { get; set; }
    public long CategoryId { get; set; }
    public Category Category { get; set; }
  }
}
```

Первым здесь добавлено свойство по имени CategoryId, являющееся примером свойства внешнего ключа, которое инфраструктура Entity Framework Core будет применять для отслеживания отношения за счет присваивания ему значения первичного ключа, идентифицирующего объект Category. Имя свойства внешнего ключа состоит из имени класса плюс имя свойства первичного ключа, давая в результате CategoryId.

Второе свойство заменяет существующее свойство Category и представляет собой пример навигационного свойства. Инфраструктура Entity Framework Core будет заполнять это свойство объектом Category, который идентифицируется свойством внешнего ключа, что делает его более подходящим для работы с данными в БД.

Обновление контекста и создание хранилища

Для обеспечения доступа к объектам Category добавьте в класс контекста БД свойство DbSet<T> (листинг 7.6).

Листинг 7.6. Добавление свойства в файле DataContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {

  public class DataContext : DbContext {

    public DataContext(DbContextOptions<DataContext> opts) : base(opts)
{}

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

    public DbSet<Category> Categories { get; set; }
  }
}
```

Новое свойство следует тому же самому шаблону, что и существующее свойство: оно объявлено как свойство public с конструкциями get и set, а возвращает экземпляр DbSet<T>, где T — класс, который нужно хранить в БД.

Когда модель данных расширяется, вы можете снабдить остальной код в приложении доступом к новым типам данных, добавив члены к существующему классу хранилища или создав новый такой класс. Для приложения SportsStore давайте создадим отдельное хранилище, просто чтобы продемонстрировать, как это делается. Добавьте в папку Models файл класса по имени CategoryRepository.cs и определите в нем интерфейс вместе с классом реализации (листинг 7.7).

Листинг 7.7. Содержимое файла CategoryRepository.cs из папки Models

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public interface ICategoryRepository {

    IEnumerable<Category> Categories { get; }

    void AddCategory(Category category);

    void UpdateCategory(Category category);

    void DeleteCategory(Category category);
  }
```

```
  public class CategoryRepository : ICategoryRepository {
    private DataContext context;

    public CategoryRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Category> Categories => context.Categories;

    public void AddCategory(Category category) {
      context.Categories.Add(category);
      context.SaveChanges();
    }

    public void UpdateCategory(Category category) {
      context.Categories.Update(category);
      context.SaveChanges();
    }

    public void DeleteCategory(Category category) {
      context.Categories.Remove(category);
      context.SaveChanges();
    }
  }
}
```

Здесь в одном файле определяются интерфейс хранилища и класс реализации, а для выполнения обновлений используется простейший подход, не полагающийся на средства обнаружения изменений. Зарегистрируйте хранилище и его реализацию в классе `Startup` для применения со средством внедрения зависимостей (листинг 7.8).

Листинг 7.8. Регистрация хранилища в файле `Startup.cs` из папки `SportsStore`

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

namespace SportsStore {
  public class Startup {

    public Startup(IConfiguration config) => Configuration = config;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddTransient<IRepository, DataRepository>();
      services.AddTransient<ICategoryRepository, CategoryRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
      options.UseSqlServer(conString));
    }
```

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

Создание и применение миграции

Инфраструктура Entity Framework Core не сможет сохранять объекты `Category` до тех пор, пока БД не будет обновлена для соответствия изменениям, внесенным в модель данных. Чтобы обновить БД, потребуется создать и применить к ней миграцию, для чего выполнить в папке проекта `SportsStore` команды из листинга 7.9.

Совет. Если при выполнении команды `dotnet ef database update` возникает исключение, то вероятная причина в том, что после выполнения команд из листинга 7.3 вы добавили в БД данные `Product`. Запустите команды из листинга 7.3 еще раз, в результате чего БД сбросится, и обновите ее миграцией, созданной в листинге 7.9.

Листинг 7.9. Создание и применение миграции к БД

```
dotnet ef migrations add Categories
dotnet ef database update
```

Первая команда создает новую миграцию по имени `Categories`, которая будет содержать команды, требующиеся для подготовки БД к хранению новых объектов. Вторая команда выполняет такие команды для обновления БД.

Создание контроллера и представления

Мы создали *обязательное* отношение между классами `Product` и `Category`, т.е. каждый объект `Product` должен быть ассоциирован с объектом `Category`. При отношении такого вида полезно снабжать пользователя средствами для управления объектами `Category` в БД. Добавьте в папку `Controllers` файл класса по имени `CategoriesController.cs` и поместите в него определение контроллера из листинга 7.10.

Совет. Альтернативой обязательному отношению является необязательное отношение, при котором объект `Product` может быть ассоциирован с объектом `Category`, но не обязан делать это. Создание отношений обоих видов подробно обсуждается в части II книги.

Листинг 7.10. Содержимое файла `CategoriesController.cs` из папки `Controllers`

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

```
namespace SportsStore.Controllers {
  public class CategoriesController : Controller {
    private ICategoryRepository repository;
    public CategoriesController(ICategoryRepository repo) =>
      repository = repo;
    public IActionResult Index() => View(repository.Categories);
    [HttpPost]
    public IActionResult AddCategory(Category category) {
      repository.AddCategory(category);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult EditCategory(long id) {
      ViewBag.EditId = id;
      return View("Index", repository.Categories);
    }
    [HttpPost]
    public IActionResult UpdateCategory(Category category) {
      repository.UpdateCategory(category);
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult DeleteCategory(Category category) {
      repository.DeleteCategory(category);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Контроллер `Categories` принимает в своем конструкторе объект хранилища для доступа к данным категорий и определяет действия, которые поддерживают запрашивание БД, а также создание, обновление и удаление объектов `Category`. Чтобы снабдить контроллер представлением, создайте папку `Views/Categories` и добавьте в нее файл по имени `Index.cshtml` с содержимым из листинга 7.11.

Листинг 7.11. Содержимое файла `Index.cshtml` из папки `Views/Categories`

```
@model IEnumerable<Category>

<h3 class="p-2 bg-primary text-white text-center">Categories</h3>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Description</div>
    <div class="col-3"></div>
  </div>
  @if (ViewBag.EditId == null) {
    <form asp-action="AddCategory" method="post">
      @Html.Partial("CategoryEditor", new Category())
    </form>
  }
```

```
  @foreach (Category c in Model) {
    @if (c.Id == ViewBag.EditId) {
      <form asp-action="UpdateCategory" method="post">
        <input type="hidden" name="Id" value="@c.Id" />
        @Html.Partial("CategoryEditor", c)
      </form>
    } else {
      <div class="row p-2">
        <div class="col-1">@c.Id</div>
        <div class="col">@c.Name</div>
        <div class="col">@c.Description</div>
        <div class="col-3">
          <form asp-action="DeleteCategory" method="post">
            <input type="hidden" name="Id" value="@c.Id" />
            <a asp-action="EditCategory" asp-route-id="@c.Id"
               class="btn btn-outline-primary">Edit</a>
            <button type="submit" class="btn btn-outline-danger">
              Delete
            </button>
          </form>
        </div>
      </div>
    }
  }
</div>
```

Представление `Index` предлагает единый интерфейс для управления категориями и поручает создание и редактирование объектов частичному представлению. Чтобы создать частичное представление, добавьте в папку `Views/Categories` файл по имени `CategoryEditor.cshtml` и поместите в него содержимое, показанное в листинге 7.12.

Листинг 7.12. Содержимое файла `CategoryEditor.cshtml` из папки `Views/Categories`

```
@model Category

<div class="row p-2">
  <div class="col-1"></div>
  <div class="col">
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="col">
    <input asp-for="Description" class="form-control" />
  </div>
  <div class="col-3">
    @if (Model.Id == 0) {
      <button type="submit" class="btn btn-primary">Add</button>
    } else {
      <button type="submit" class="btn btn-outline-primary">Save</button>
      <a asp-action="Index" class="btn btn-outline-secondary">Cancel</a>
    }
  </div>
</div>
```

Для облечения перемещения по приложению добавьте к разделяемой компоновке элементы, приведенные в листинге 7.13.

Листинг 7.13. Добавление элементов в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <style>
    .placeholder { visibility: collapse; display: none }
    .placeholder:only-child { visibility: visible; display: flex }
  </style>
</head>
<body>
  <div class="container-fluid">
    <div class="row p-2">
      <div class="col-2">
        <a asp-controller="Home" asp-action="Index"
           class="@GetClassForButton("Home")">
          Products
        </a>
        <a asp-controller="Categories" asp-action="Index"
           class="@GetClassForButton("Categories")">
          Categories
        </a>
      </div>
      <div class="col">
        @RenderBody()
      </div>
    </div>
  </div>
</body>
</html>

@functions {
  string GetClassForButton(string controller) {
    return "btn btn-block " + (ViewContext.RouteData.Values["controller"]
      as string == controller ? "btn-primary" : "btn-outline-primary");
  }
}
```

В разделяемую компоновку добавлены кнопки, выбирающие контроллеры `Product` и `Categories` с помощью простой встроенной функции, которая использует CSS-стили Bootstrap для выделения кнопки, соответствующей отображаемому в текущий момент контроллеру.

На заметку! Я не часто применяю встроенные функции Razor, поскольку предпочитаю удерживать весь код C# в файлах классов. Но в этом случае функция обладает преимуществом сохранения примера кратким и относится только к содержимому в представлении, вдобавок создать ее проще, чем компонент представления.

Заполнение базы данных категориями

Завершая отношение между данными, удобно иметь какие-то данные для работы с ними. Запустите приложение, используя `dotnet run`, щелкните на кнопке Categories (Категории) и с применением HTML-формы создайте категории согласно значениям в табл. 7.1.

Таблица 7.1. Значения данных для создания категорий

Name (Название)	Description (Описание)
`Watersports` (Водный спорт)	`Make a splash` (Наделайте шуму)
`Soccer` (Футбол)	`The world's favorite game` (Любимая игра в мире)
`Running` (Бег)	`Run like the wind` (Пробежитесь с ветерком)

После добавления всех трех категорий вы должны увидеть содержимое, показанное на рис. 7.2.

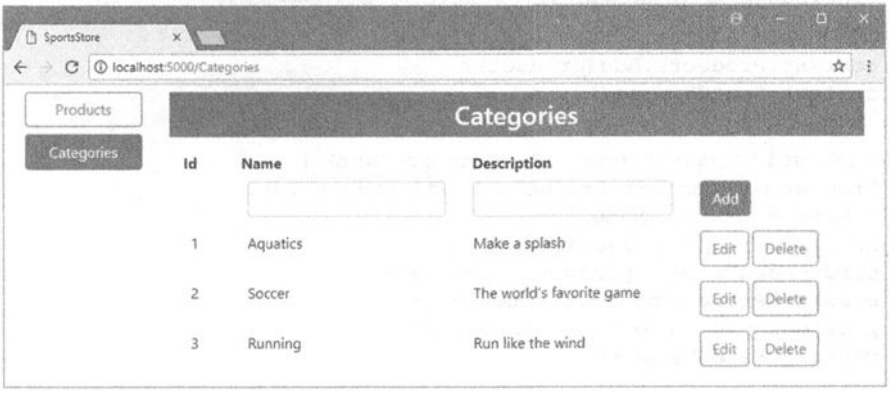

Рис. 7.2. Добавление данных о категориях в приложение

Использование отношения между данными

Часть приложения, которая имеет дело с объектами `Product`, потребуется обновить для отражения нового отношения в БД. Процесс предусматривает решение двух задач: включение данных о категориях при запрашивании БД и предоставление пользователю возможности выбрать категорию при создании или редактировании сведений о товаре.

Работа со связанными данными

Инфраструктура Entity Framework Core игнорирует отношения, если только они явно не включаются в запросы. Это означает, что навигационные свойства, такие как свойство `Category`, определенное в классе `Product`, по умолчанию будут оставляться равными `null`. Расширяющий метод `Include()` применяется для сообщения инфраструктуре Entity Framework Core о необходимости заполнения навигационного свойства связанными данными и вызывается на объекте реализации `IQueryable<T>`,

который представляет запрос. В листинге 7.14 метод `Include()` используется для включения связанных объектов `Category` в запросы, сделанные хранилищем товаров.

Листинг 7.14. Включение связанных данных в файле `DataRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products
      .Include(p => p.Category).ToArray();

    public Product GetProduct(long key) => context.Products
      .Include(p => p.Category).First(p => p.Id == key);

    public void AddProduct(Product product) {
      context.Products.Add(product);
      context.SaveChanges();
    }

    public void UpdateProduct(Product product) {
      Product p = context.Products.Find(product.Id);
      p.Name = product.Name;
      // p.Category = product.Category;
      p.PurchasePrice = product.PurchasePrice;
      p.RetailPrice = product.RetailPrice;
      p.CategoryId = product.CategoryId;
      context.SaveChanges();
    }

    public void UpdateAll(Product[] products) {
      Dictionary<long, Product> data = products.ToDictionary(p => p.Id);
      IEnumerable<Product> baseline =
        context.Products.Where(p => data.Keys.Contains(p.Id));
      foreach(Product databaseProduct in baseline) {
        Product requestProduct = data[databaseProduct.Id];
        databaseProduct.Name = requestProduct.Name;
        databaseProduct.Category = requestProduct.Category;
        databaseProduct.PurchasePrice = requestProduct.PurchasePrice;
        databaseProduct.RetailPrice = requestProduct.RetailPrice;
      }
      context.SaveChanges();
    }

    public void Delete(Product product) {
      context.Products.Remove(product);
      context.SaveChanges();
    }
  }
}
```

Метод `Include()` определен в пространстве имен `Microsoft.EntityFrameworkCore` и принимает лямбда-выражение, выбирающее навигационное свойство, которое желательно включить в запрос. Метод `Find()`, применяемый для метода `GetProduct()`, не может использоваться с методом `Include()`, поэтому он заменен методом `First()`, который приводит к тому же самому эффекту. Результат внесенных изменений заключается в том, что инфраструктура Entity Framework Core будет заполнять навигационное свойство `Product.Category` для объектов `Product`, созданных свойством `Products` и методом `GetProduct()`.

Обратите внимание на изменения в методе `UpdateProduct()`. Во-первых, исходные данные запрашиваются напрямую, а не через метод `GetProduct()`, потому что загружать связанные данные при выполнении обновления нежелательно. Во-вторых, закомментирован оператор, устанавливающий свойство `Category`, и добавлен оператор, который взамен устанавливает свойство `CategoryId`. Установка свойства внешнего ключа — все, что необходимо инфраструктуре Entity Framework Core для обновления отношения между двумя объектами в БД.

Выбор категории для товара

Обновите контроллер `Home`, чтобы он имел доступ к данным `Category` через хранилище и передавал их своему представлению (листинг 7.15). Это позволит представлению предложить выбор из полного набора категорий при редактировании или создании объекта `Product`.

Листинг 7.15. Применение данных о категориях в файле `HomeController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
  public class HomeController : Controller {
    private IRepository repository;
    private ICategoryRepository catRepository;
    public HomeController(IRepository repo, ICategoryRepository catRepo) {
      repository = repo;
      catRepository = catRepo;
    }
    public IActionResult Index() {
      return View(repository.Products);
    }
    public IActionResult UpdateProduct(long key) {
      ViewBag.Categories = catRepository.Categories;
      return View(key == 0 ? new Product() : repository.GetProduct(key));
    }
    [HttpPost]
    public IActionResult UpdateProduct(Product product) {
      if (product.Id == 0) {
        repository.AddProduct(product);
      } else {
        repository.UpdateProduct(product);
      }
      return RedirectToAction(nameof(Index));
    }
```

```
    [HttpPost]
    public IActionResult Delete(Product product) {
      repository.Delete(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы предоставить пользователю возможность выбора одной из категорий при создании или редактировании объекта `Product`, добавьте в представление `UpdateProduct` элемент `select` (листинг 7.16).

Листинг 7.16. Отображение категорий в файле `UpdateProduct.html` из папки `Views/Home`

```
@model Product

<h3 class="p-2 bg-primary text-white text-center">Update Product</h3>

<form asp-action="UpdateProduct" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" readonly />
  </div>
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Category"></label>
    <select class="form-control" asp-for="CategoryId">
      @if (Model.Id == 0) {
        <option disabled selected>Choose Category</option>
      }
      @foreach (Category c in ViewBag.Categories) {
        <option selected=@(Model.Category?.Id == c.Id)
              value="@c.Id">@c.Name</option>
      }
    </select>
  </div>
  <div class="form-group">
    <label asp-for="PurchasePrice"></label>
    <input asp-for="PurchasePrice" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="RetailPrice"></label>
    <input asp-for="RetailPrice" 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 разметку включен элемент-заполнитель `option` на случай, если представление используется для создания нового объекта `Product`, и предусмотрено выражение Razor для применения атрибута `selected`, если текущий объект редактируется.

Осталось лишь обновить представление `Index`, чтобы проследовать по навигационному свойству и отобразить для каждого объекта `Product` название выбранной категории (листинг 7.17).

Листинг 7.17. Следование по навигационному свойству в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>

  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col-1">@p.Id</div>
      <div class="col">@p.Name</div>
      <div class="col">@p.Category.Name</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
      <div class="col">
        <form asp-action="Delete" method="post">
          <a asp-action="UpdateProduct" asp-route-key="@p.Id"
             class="btn btn-outline-primary">
            Edit
          </a>
          <input type="hidden" name="Id" value="@p.Id" />
          <button type="submit" class="btn btn-outline-danger">
            Delete
          </button>
        </form>
      </div>
    </div>
  }

  <div class="text-center p-2">
    <a asp-action="UpdateProduct" asp-route-key="0"
       class="btn btn-primary">Add</a>
  </div>
</div>
```

Создание и редактирование товаров с категориями

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Add (Добавить) и примените HTML-форму для создания объектов `Product` согласно данным в табл. 7.2. При создании каждого объекта выбирайте категорию из списка в элементе `select`.

Таблица 7.2. Значения данных для создания тестовых объектов `Product`

Name (Наименование)	Category (Категория)	Purchase Price (Покупная цена)	Retail Price (Розничная цена)
Kayak (Каяк)	Watersports (Водный спорт)	200	275
Lifejacket (Спасательный жилет)	Watersports (Водный спорт)	30	48.95
Soccer Ball (Футбольный мяч)	Soccer (Футбол)	17	19.50

После создания каждого объекта будет инициироваться действие `Index` для отображения результатов, которое заставит инфраструктуру Entity Framework Core запросить в БД данные `Product` и связанные с ними объекты `Category`. Вы можете увидеть, как он транслируется в SQL-запрос, просмотрев сгенерированные приложением журнальные сообщения:

```
...
SELECT [p].[Id], [p].[CategoryId], [p].[Name], [p].[PurchasePrice],
  [p].[RetailPrice], [p.Category].[Id], [p.Category].[Description],
  [p.Category].[Name]
FROM [Products] AS [p]
INNER JOIN [Categories] AS [p.Category] ON [p].[CategoryId] =
[p.Category].[Id]
...
```

Инфраструктура Entity Framework Core использует внешний ключ для запрашивания данных, которые необходимы для создания объектов `Category`, связанных с объектами `Product`, и применяет внутреннее соединение для объединения данных из таблиц `Products` и `Categories`.

После создания всех трех объектов `Product` щелкните на кнопке Categories (Категории), щелкните на кнопке Edit (Редактировать) для категории `Watersports` и измените значение в поле Name (Наименование) на `Aquatics` (Водные виды спорта). Щелкните на кнопке Save (Сохранить), затем на кнопке Products (Товары) и вы заметите, что оба объекта `Product` в отредактированной категории отображаются с новым наименованием (рис. 7.3).

Внимание! Если вы удалите объект `Category`, то связанные с ним объекты `Product` также удалятся, что является стандартной конфигурацией для обязательных отношений. Стандартная конфигурация и альтернативные варианты конфигурации более подробно обсуждаются в главе 22.

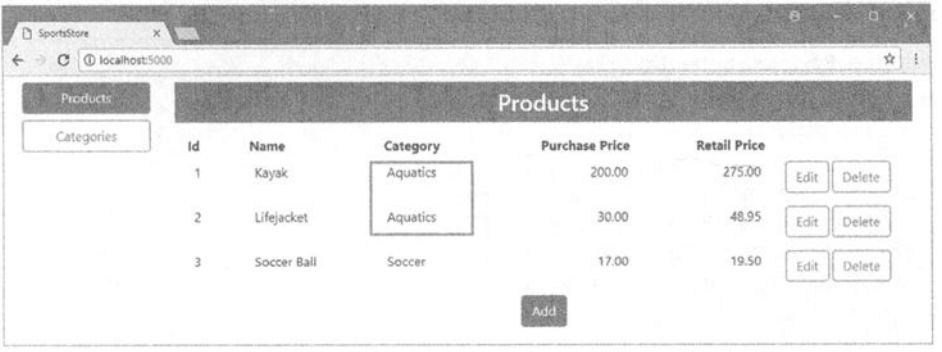

Рис. 7.3. Результат изменения наименования категории

Добавление поддержки для заказов

Чтобы продемонстрировать более сложное отношение, мы добавим поддержку для создания и сохранения заказов и будем использовать их для представления выбора товаров, произведенного покупателями. В последующих разделах мы расширим модель данных дополнительными классами, обновим БД и добавим контроллер для управления новыми данными.

Создание классов модели данных

Начните с добавления в папку `Models` файла по имени `Order.cs` с определением класса, показанным в листинге 7.18.

Листинг 7.18. Содержимое файла `Order.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {

  public class Order {
    public long Id { get; set; }

    public string CustomerName { get; set; }
    public string Address { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
    public bool Shipped { get; set; }

    public IEnumerable<OrderLine> Lines { get; set; }
  }
}
```

Класс `Order` имеет свойства, которые хранят имя и адрес покупателя, а также признак, доставлены ли товары. Есть также навигационное свойство, обеспечивающее доступ к связанным объектам `OrderLine`, которые будут представлять отдельные выбранные товары. Для создания класса `OrderLine` добавьте в папку `Models` файл по имени `OrderLine.cs` с кодом из листинга 7.19.

Листинг 7.19. Содержимое файла `OrderLine.cs` из папки `Models`

```
namespace SportsStore.Models {
  public class OrderLine {
    public long Id { get; set; }
    public long ProductId { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
    public long OrderId { get; set; }
    public Order Order { get; set; }
  }
}
```

Каждый объект `OrderLine` связан с объектами `Order` и `Product` и имеет свойство, которое отражает, сколько товара покупатель заказал. Чтобы обеспечить удобный доступ к данным `Order`, добавьте в класс контекста свойства, приведенные в листинге 7.20.

Листинг 7.20. Добавление свойств в файле `DataContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
  public class DataContext : DbContext {
    public DataContext(DbContextOptions<DataContext> opts) : base(opts) {}
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderLine> OrderLines { get; set; }
  }
}
```

Создание хранилища и подготовка базы данных

Для предоставления согласованного доступа к новым данным остальному коду приложения добавьте в папку `Models` файл по имени `IOrdersRepository.cs` с определением интерфейса, показанным в листинге 7.21.

Листинг 7.21. Содержимое файла `IOrdersRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace SportsStore.Models {
  public interface IOrdersRepository {
    IEnumerable<Order> Orders { get; }
    Order GetOrder(long key);
    void AddOrder(Order order);
    void UpdateOrder(Order order);
    void DeleteOrder(Order order);
  }
}
```

Добавьте в папку Models файл по имени OrdersRepository.cs с классом реализации, представленным в листинге 7.22.

Листинг 7.22. Содержимое файла OrdersRepository.cs из папки Models

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

namespace SportsStore.Models {

  public class OrdersRepository : IOrdersRepository {
    private DataContext context;

    public OrdersRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Order> Orders => context.Orders
      .Include(o => o.Lines).ThenInclude(l => l.Product);

    public Order GetOrder(long key) => context.Orders
      .Include(o => o.Lines).First(o => o.Id == key);

    public void AddOrder(Order order) {
      context.Orders.Add(order);
      context.SaveChanges();
    }

    public void UpdateOrder(Order order) {
      context.Orders.Update(order);
      context.SaveChanges();
    }

    public void DeleteOrder(Order order) {
      context.Orders.Remove(order);
      context.SaveChanges();
    }
  }
}
```

Реализация хранилища следует шаблону, принятому для других хранилищ, и ради простоты не задействует средство обнаружения изменений. Обратите внимание на применение методов Include() и ThenInclude() для навигации по модели данных и добавления в запросы связанных данных — процесс, который подробно описан в главах 14–16. Добавьте в класс Startup оператор, чтобы система внедрения зависимостей распознавала зависимости от интерфейса IOrderRepository с использованием кратковременных объектов OrderRepository (листинг 7.23).

Листинг 7.23. Конфигурирование внедрения зависимостей 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.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace SportsStore {
  public class Startup {

    public Startup(IConfiguration config) => Configuration = config;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddTransient<IRepository, DataRepository>();
      services.AddTransient<ICategoryRepository, CategoryRepository>();
      services.AddTransient<IOrdersRepository, OrdersRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
        options.UseSqlServer(conString));
    }

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

Запустите в папке проекта `SportsStore` команды, показанные в листинге 7.24, чтобы подготовить БД к хранению новых классов модели данных путем создания и применения миграции Entity Framework Core.

Листинг 7.24. Создание и применение новой миграции БД

```
dotnet ef migrations add Orders
dotnet ef database update
```

Создание контроллеров и представлений

Весь связующий код Entity Framework Core для работы с объектами `Order` на месте; следующий шаг предусматривает добавление средств MVC, которые позволят создавать и управлять экземплярами. Добавьте в папку `Controllers` файл класса по имени `OrdersController.cs` и определите в нем контроллер (листинг 7.25). Реализация метода `AddOrUpdateOrder()` будет завершена, когда станут доступными остальные средства.

Листинг 7.25. Содержимое файла `OrdersController.cs` из папки `Controllers`

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

```
namespace SportsStore.Controllers {
  public class OrdersController : Controller {
    private IRepository productRepository;
    private IOrdersRepository ordersRepository;

    public OrdersController(IRepository productRepo,
          IOrdersRepository orderRepo) {
      productRepository = productRepo;
      ordersRepository = orderRepo;
    }

    public IActionResult Index() => View(ordersRepository.Orders);

    public IActionResult EditOrder(long id) {
      var products = productRepository.Products;
      Order order = id == 0 ? new Order() : ordersRepository.GetOrder(id);
      IDictionary<long, OrderLine> linesMap
        = order.Lines?.ToDictionary(l => l.ProductId)
        ?? new Dictionary<long, OrderLine>();
      ViewBag.Lines = products.Select(p => linesMap.ContainsKey(p.Id)
        ? linesMap[p.Id]
        : new OrderLine { Product = p, ProductId = p.Id, Quantity = 0 });
      return View(order);
    }

    [HttpPost]
    public IActionResult AddOrUpdateOrder(Order order) {

      // ...метод действия, подлежащий завершению...

      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult DeleteOrder(Order order) {
      ordersRepository.DeleteOrder(order);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Операторы LINQ в методе действия `EditOrder()` могут выглядеть запутанными, но они осуществляют подготовку данных `OrderLine`, чтобы иметь один объект `OrderLine` для каждого объекта `Product`, даже если ранее этот товар не выбирался.

Таким образом, для нового заказа свойство `ViewBag.Lines` будет заполняться последовательностью объектов `OrderLine`, соответствующих каждому объекту `Product` в БД, с установленными в `0` свойствами `Id` и `Quantity`. Когда объект сохраняется в БД, нулевое значение `Id` будет указывать, что объект является новым, и сервер баз данных присвоит ему новый уникальный первичный ключ.

Для существующих заказов свойство `ViewBag.Lines` будет заполняться объектами `OrderLine`, прочитанными из БД, и дополнительными объектами с нулевыми свойствами `Id` для остальных товаров.

Такая структура извлекает преимущество из способа совместной работы ASP.NET Core MVC и Entity Framework Core и упрощает процесс обновления БД, как вы увидите при реализации остатка примера.

Далее нужно создать представление, которое будет отображать список всех объектов в БД. Добавьте папку `Views/Orders` и поместите в нее файл по имени `Index.cshtml` с содержимым, приведенным в листинге 7.26.

Листинг 7.26. Содержимое файла `Index.cshtml` из папки `Views/Orders`

```
@model IEnumerable<Order>

<h3 class="p-2 bg-primary text-white text-center">Orders</h3>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Zip</div>
    <div class="col font-weight-bold">Total</div>
    <div class="col font-weight-bold">Profit</div>
    <div class="col-1 font-weight-bold">Status</div>
    <div class="col-3"></div>
  </div>
  <div>
    <div class="row placeholder p-2"><div class="col-12 text-center">
      <h5>No Orders</h5>
    </div></div>
    @foreach (Order o in Model) {
      <div class="row p-2">
        <div class="col-1">@o.Id</div>
        <div class="col">@o.CustomerName</div>
        <div class="col">@o.ZipCode</div>
        <div class="col">@o.Lines.Sum(l => l.Quantity
          * l.Product.RetailPrice)</div>
        <div class="col">@o.Lines.Sum(l => l.Quantity
          * (l.Product.RetailPrice - l.Product.PurchasePrice))</div>
        <div class="col-1">@(o.Shipped ? "Shipped" : "Pending")</div>
        <div class="col-3 text-right">
          <form asp-action="DeleteOrder" method="post">
            <input type="hidden" name="Id" value="@o.Id" />
            <a asp-action="EditOrder" asp-route-id="@o.Id"
               class="btn btn-outline-primary">Edit</a>
            <button type="submit" class="btn btn-outline-danger">
              Delete
            </button>
          </form>
        </div>
      </div>
    }
  </div>
</div>
<div class="text-center">
  <a asp-action="EditOrder" class="btn btn-primary">Create</a>
</div>
```

Представление отображает сводку по объектам `Order` в БД, а также суммарную стоимость заказанных товаров и размер прибыли, которая будет получена. Здесь

присутствуют кнопки для создания нового заказа и для редактирования и удаления существующего заказа.

Чтобы снабдить приложение представлением для создания или редактирования заказа, добавьте в папку `Views/Orders` файл по имени `EditOrder.cshtml` и поместите в него содержимое из листинга 7.27.

Листинг 7.27. Содержимое файла `EditOrder.cshtml` из папки `Views/Orders`

```
@model Order

<h3 class="p-2 bg-primary text-white text-center">Create/Update Order</h3>

<form asp-action="AddOrUpdateOrder" method="post">
  <div class="form-group">
    <label asp-for="Id"></label>
    <input asp-for="Id" class="form-control" readonly />
  </div>
  <div class="form-group">
    <label asp-for="CustomerName"></label>
    <input asp-for="CustomerName" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="Address"></label>
    <input asp-for="Address" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="State"></label>
    <input asp-for="State" class="form-control" />
  </div>
  <div class="form-group">
    <label asp-for="ZipCode"></label>
    <input asp-for="ZipCode" class="form-control" />
  </div>
  <div class="form-check">
    <label class="form-check-label">
      <input type="checkbox" asp-for="Shipped" class="form-check-input" />
      Shipped
    </label>
  </div>
  <h6 class="mt-1 p-2 bg-primary text-white text-center">Products
Ordered</h6>
  <div class="container-fluid">
    <div class="row">
      <div class="col font-weight-bold">Product</div>
      <div class="col font-weight-bold">Category</div>
      <div class="col font-weight-bold">Quantity</div>
    </div>
    @{ int counter = 0; }
    @foreach (OrderLine line in ViewBag.Lines) {
      <input type="hidden" name="lines[@counter].Id" value="@line.Id" />
      <input type="hidden" name="lines[@counter].ProductId"
            value="@line.ProductId" />
      <input type="hidden" name="lines[@counter].OrderId" value="@Model.Id" />
      <div class="row mt-1">
```

```
        <div class="col">@line.Product.Name</div>
        <div class="col">@line.Product.Category.Name</div>
        <div class="col">
          <input type="number" name="lines[@counter].Quantity"
                value="@line.Quantity" />
        </div>
      </div>
      counter++;
    }
  </div>
  <div class="text-center m-2">
    <button type="submit" class="btn btn-primary">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Представление `EditOrder` предлагает пользователю форму с элементами `input` для свойств, определенных в классе `Order`, и элементами для всех объектов `Product` в БД, которые будут заполняться заказанным количеством при редактировании существующих объектов.

Для облечения доступа к новым средствам добавьте в компоновку, разделяемую всеми представлениями, элемент из листинга 7.28.

Листинг 7.28. Добавление элемента в файле `_Layout.cshtml` из папки `Views/Shared`

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <style>
    .placeholder { visibility: collapse; display: none }
    .placeholder:only-child { visibility: visible; display: flex }
  </style>
</head>
<body>
  <div class="container-fluid">
    <div class="row p-2">
      <div class="col-2">
        <a asp-controller="Home" asp-action="Index"
           class="@GetClassForButton("Home")">
          Products
        </a>
        <a asp-controller="Categories" asp-action="Index"
           class="@GetClassForButton("Categories")">
          Categories
        </a>
        <a asp-controller="Orders" asp-action="Index"
           class="@GetClassForButton("Orders")">
          Orders
        </a>
      </div>
```

```
      <div class="col">
        @RenderBody()
      </div>
    </div>
  </div>
</body>
</html>

@functions {
  string GetClassForButton(string controller) {
    return "btn btn-block " + (ViewContext.RouteData.Values["controller"]
      as string == controller ? "btn-primary" : "btn-outline-primary");
  }
}
```

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Orders (Заказы) и затем на кнопке Create (Создать). Вы увидите пустую форму, которая содержит элементы для всех товаров в БД. Поскольку создается новый заказ, во всех полях Quantity (Количество) находятся нули (рис. 7.4).

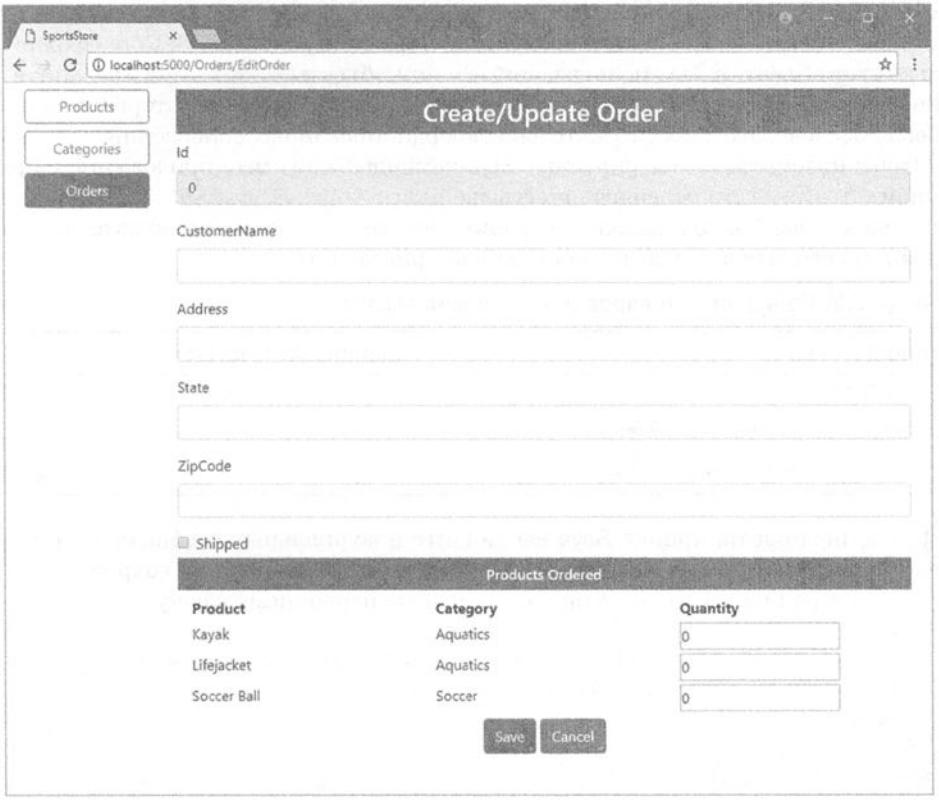

Рис. 7.4. Представление для создания нового заказа

Сохранение данных заказа

Щелчок на кнопке Save не приводит к сохранению каких-либо данных, потому что метод `AddOrUpdateOrder()` в листинге 7.25 остался незавершенным. Чтобы закончить контроллер, добавьте в этот метод действия код, показанный в листинге 7.29.

Листинг 7.29. Сохранение данных в файле `OrdersController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult AddOrUpdateOrder(Order order) {
  order.Lines = order.Lines
      .Where(l => l.Id > 0 || (l.Id == 0 && l.Quantity > 0)).ToArray();
  if (order.Id == 0) {
    ordersRepository.AddOrder(order);
  } else {
    ordersRepository.UpdateOrder(order);
  }
  return RedirectToAction(nameof(Index));
}
...
```

Операторы в методе действия полагаются на удобную функциональную особенность Entity Framework Core: в случае передачи объекта `Order` методу `AddOrder()` или `UpdateOrder()` хранилища инфраструктура Entity Framework Core сохранит не только этот объект `Order`, но и связанные с ним объекты `OrderLine`. Указанная особенность может не выглядеть важной, но она упрощает процесс, который иначе потребовал последовательности тщательно скоординированных обновлений.

Чтобы просмотреть генерируемые SQL-команды, запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000/orders`, щелкните на кнопке Create и заполните форму. Детали, касающиеся покупателя, роли не играют, но вводите количества для товаров, приведенные в табл. 7.3.

Таблица 7.3. Количества товаров для создания заказа

Product (Товар)	Quantity (Количество)
`Kayak` (Каяк)	1
`Lifejacket` (Спасательный жилет)	2
`Soccer Ball` (Футбольный мяч)	0

После щелчка на кнопке Save вы увидите в журнальных сообщениях, сгенерированных приложением, несколько SQL-команд. Первая команда сохраняет объект `Order`, а вторая получает значение, назначенное первичному ключу:

```
...
INSERT INTO [Orders] ([Address], [CustomerName], [Shipped], [State],
[ZipCode]) VALUES (@p0, @p1, @p2, @p3, @p4);

SELECT [Id]
FROM [Orders]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

Далее инфраструктура Entity Framework Core использует первичный ключ объекта `Order` для сохранения объектов `OrderLine`:

```
...
DECLARE @inserted0 TABLE ([Id] bigint, [_Position] [int]);
MERGE [OrderLines] USING (
VALUES (@p5, @p6, @p7, 0),
  (@p8, @p9, @p10, 1)) AS i ([OrderId], [ProductId], [Quantity], _
Position) ON 1=0
WHEN NOT MATCHED THEN
  INSERT ([OrderId], [ProductId], [Quantity])
  VALUES (i.[OrderId], i.[ProductId], i.[Quantity])
  OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;
...
```

Наконец, Entity Framework Core запрашивает у БД первичные ключи, назначенные объектам `OrderLine`:

```
...
SELECT [t].[Id] FROM [OrderLines] t
INNER JOIN @inserted0 i ON ([t].[Id] = [i].[Id])
ORDER BY [i].[_Position];
...
```

Не переживайте, если вы не смогли полностью вникнуть в смысл показанных выше SQL-команд. Просто важно понимать, что инфраструктура Entity Framework Core заботится о сохранении связанных данных и автоматически генерирует SQL-команды. Последний момент, который нужно отметить в листинге 7.29, касается следующего оператора:

```
...
order.Lines = order.Lines
     .Where(l => l.Id > 0 || (l.Id == 0 && l.Quantity > 0)).ToArray();
...
```

Оператор исключает любой объект `OrderLine`, для которого не было выбрано ненулевое количество, кроме объектов, уже хранящихся в БД. Это гарантирует, что БД не переполнится объектами `OrderLine`, которые не являются частью какого-то заказа, но разрешает вносить изменения в ранее сохраненные данные. После сохранения данных отобразится сводка по заказу, как показано на рис. 7.5.

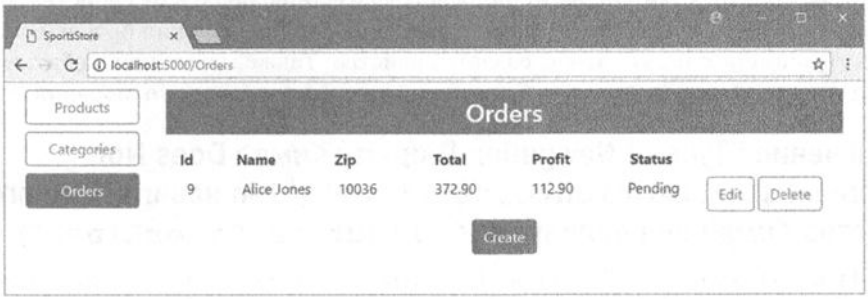

Рис. 7.5. Сводка по заказу

Распространенные проблемы и их решения

Средства для создания и работы со связанными данными могут вызывать затруднения, поэтому в последующих разделах будут описаны распространенные проблемы и даны объяснения, как их решать.

Исключение "ALTER TABLE Conflicted With The FOREIGN KEY" ("ALTER TABLE конфликтует с FOREIGN KEY")

Это исключение обычно возникает, когда вы применяете миграцию к БД, содержащей ограничения, с которыми не согласуются существующие данные в БД. Например, вы получите такое исключение в приложении SportsStore, если создали объекты Product в БД до применения миграции, которая добавляет поддержку для отдельных объектов Category. Существующие в БД данные не удовлетворяют требованию иметь отношение внешнего ключа с категорией, а потому обновление БД потерпит неудачу.

Простейший способ решения проблемы предусматривает удаление и воссоздание БД, что приведет к удалению любых содержащихся в ней данных. Однако такой подход неприемлем в производственной системе, где данные должны быть аккуратно модифицированы, прежде чем можно будет применить миграцию.

Исключение "UPDATE Conflicted With The FOREIGN KEY" ("UPDATE конфликтует с FOREIGN KEY")

Это исключение возникает, когда вы пытаетесь сохранить новый объект или обновить существующий, используя данные, которые не согласуются с ограничениями, примененными к БД для поддержки связанных данных. Наиболее вероятной причиной является отсутствие значения ключа для связанных данных. Скажем, в контексте приложения SportsStore такое исключение произойдет, если вы попробуете сохранить или обновить объект Product, не указав значение для свойства CategoryId. Если вы получили подобное исключение, тогда удостоверьтесь в том, что HTML-форма содержит элементы input для всех свойств внешних ключей, определяемых классами модели данных.

Исключение "The Property Expression 'x => x.<имя>' is Not Valid" ("Выражение свойства x => x.<имя> является недопустимым")

Такое исключение возникает, когда вы забыли добавить к навигационному свойству конструкции get и set и затем выбираете его с использованием метода Include(). Отсутствие конструкций get и set приводит к созданию поля, а не свойства, и метод Include() не может проследовать по нему в запросе. Для решения проблемы добавьте конструкции get и set, чтобы создать свойство. Также может понадобиться заново создать и повторно применить миграцию, которая определяет отношение.

Исключение "Type of Navigation Property <имя> Does Not Implement ICollection<OrderLine>" ("Тип навигационного свойства <имя> не реализует ICollection<OrderLine>")

Инфраструктура Entity Framework Core чувствительна к типам данных, назначенным навигационным свойствам, и это исключение может возникнуть при выполнении запроса LINQ на коллекции навигационных свойств перед сохранением данных, как делалось в листинге 7.29 для приложения SportsStore. Чтобы устранить проблему, вызовите метод ToArray() на результате запроса LINQ, что выпустит результат, который реализует интерфейс, ожидаемый Entity Framework Core.

Исключение "The Property <*имя*> is Not a Navigation Property of Entity Type <*имя*>" ("Свойство <*имя*> не является навигационным свойством сущностного типа <*имя*>")

Такое исключение происходит, когда вы используете метод `Include()` для выбора свойства, по которому инфраструктура Entity Framework Core не может проследовать для получения доступа к связанным данным. Самой распространенной причиной проблемы является выбор свойства внешнего ключа вместо навигационного свойства, с которым оно взаимодействует. В случае приложения `SportsStore` вы столкнетесь с этой ошибкой, если примените метод `Include()` для выбора из объекта `OrderLine` свойства `ProductId`, а не свойства `Product`.

Исключение "Invalid Object Name <*имя*>" ("Недопустимое имя объекта <*имя*>")

Такое исключение обычно возникает, когда вы создали миграцию, которая расширяет модель данных отношением, но забыли применить ее к БД. Используйте команду `dotnet ef database update` для применения миграций в своем проекте; за дополнительными сведениями о работе миграций обращайтесь в главу 13.

Вместо того чтобы обновляться, объекты удаляются

Если вы обнаруживаете, что попытка обновления объекта в действительности приводит к его удалению, тогда вероятная причина состоит в том, что вы загружаете связанные данные и затем устанавливаете навигационное свойство в `null` при получении исходных данных для обнаружения изменений. Например, в случае приложения `SportsStore` вы можете столкнуться с таким поведением путем запрашивания исходных данных для объекта `Product`, использования метода `Include()` для загрузки связанного объекта `Category` и установки свойства `Category` в `null` перед вызовом метода `SaveChanges()`. Описанное сочетание действий приводит к тому, что инфраструктура Entity Framework Core тихо удаляет объект, который вы намеревались обновить. Чтобы исправить проблему, не загружайте связанные данные и не устанавливайте навигационное свойство в `null`.

В представлении отображается имя класса для связанных данных

Эта проблема вызвана применением выражения Razor, которое выбирает значение навигационного свойства, возвращающего связанный объект. Затем Razor вызывает метод `ToString()`, который возвращает имя класса. Для включения в представление значения из связанных данных выберите одно из свойств связанного объекта, чтобы использовать `@Category.Name`, а не просто `@Category`, например.

Резюме

В главе модель данных `SportsStore` была расширена за счет добавления новых классов и создания отношений между ними. Вы узнали, как запрашивать связанные данные, каким образом выполнять обновления и как решать наиболее распространенные проблемы, которые могут возникать с новыми средствами. В следующей главе будет показано, каким образом приспособить части MVC и Entity Framework Core к работе с крупными объемами данных.

ГЛАВА 8

SportsStore: масштабирование

При создании приложения основное внимание обычно уделяется построению правильного фундамента и как раз такой подход был принят в проекте SportsStore. По мере развития приложения зачастую полезно увеличивать объем данных, с которыми производится работа, чтобы можно было видеть, какое влияние они оказывают на операции, инициируемые пользователем, и на время, требующееся для их выполнения. В этой главе в БД будут добавлены тестовые данные, чтобы продемонстрировать недостатки способа, которым приложение SportsStore представляет данные пользователю, и принять соответствующие меры за счет добавления поддержки разбиения на страницы, упорядочения и поиска в данных. Также будет показано, каким образом улучшить производительность операций с использованием инфраструктуры Entity Framework Core, которая поддерживает расширенные варианты конфигурации модели данных, известные как интерфейс Fluent API.

Подготовительные шаги

Мы продолжим использовать проект SportsStore, который был создан в главе 4 и модифицирован в последующих главах. Запустите в папке проекта SportsStore команды из листинга 8.1, чтобы удалить и заново создать БД.

Совет. Проект SportsStore и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Листинг 8.1. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Создание контроллера и представления для начального заполнения данными

Нам необходим контроллер, который сможет заполнять БД тестовыми данными. Добавьте в папку Controllers файл по имени SeedController.cs и определите в нем контроллер, как показано в листинге 8.2.

Листинг 8.2. Содержимое файла `SeedController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  public class SeedController : Controller {
    private DataContext context;

    public SeedController(DataContext ctx) => context = ctx;

    public IActionResult Index() {
      ViewBag.Count = context.Products.Count();
      return View(context.Products
        .Include(p => p.Category).OrderBy(p => p.Id).Take(20));
    }

    [HttpPost]
    public IActionResult CreateSeedData(int count) {
      ClearData();
      if (count > 0) {
        context.Database.SetCommandTimeout(System.TimeSpan.FromMinutes(10));
        context.Database
          .ExecuteSqlCommand("DROP PROCEDURE IF EXISTS CreateSeedData");
        context.Database.ExecuteSqlCommand($@"
          CREATE PROCEDURE CreateSeedData
            @RowCount decimal
          AS
            BEGIN
              SET NOCOUNT ON
              DECLARE @i INT = 1;
              DECLARE @catId BIGINT;
              DECLARE @CatCount INT = @RowCount / 10;
              DECLARE @pprice DECIMAL(5,2);
              DECLARE @rprice DECIMAL(5,2);
              BEGIN TRANSACTION
                WHILE @i <= @CatCount
                  BEGIN
                    INSERT INTO Categories (Name, Description)
                    VALUES (CONCAT('Category-', @i),
                      'Test Data Category');
                    SET @catId = SCOPE_IDENTITY();
                    DECLARE @j INT = 1;
                    WHILE @j <= 10
                      BEGIN
                        SET @pprice = RAND()*(500-5+1);
                        SET @rprice = (RAND() * @pprice)
                          + @pprice;
                        INSERT INTO Products (Name, CategoryId,
                          PurchasePrice, RetailPrice)
                        VALUES (CONCAT('Product', @i, '-', @j),
                          @catId, @pprice, @rprice)
                        SET @j = @j + 1
                      END
```

```
                SET @i = @i + 1
              END
            COMMIT
          END");
      context.Database.BeginTransaction();
      context.Database
        .ExecuteSqlCommand($"EXEC CreateSeedData @RowCount = {count}");
      context.Database.CommitTransaction();
    }
    return RedirectToAction(nameof(Index));
  }

  [HttpPost]
  public IActionResult ClearData() {
    context.Database.SetCommandTimeout(System.TimeSpan.FromMinutes(10));
    context.Database.BeginTransaction();
    context.Database.ExecuteSqlCommand("DELETE FROM Orders");
    context.Database.ExecuteSqlCommand("DELETE FROM Categories");
    context.Database.CommitTransaction();
    return RedirectToAction(nameof(Index));
  }
 }
}
```

Когда дело доходит до генерации крупных объемов тестовых данных, то создание объектов .NET и их сохранение в БД будет неэффективным. Контроллер `Seed` применяет средства Entity Framework Core для работы напрямую с SQL, чтобы создать и выполнить хранимую процедуру, которая производит тестовые данные гораздо быстрее. (Такие средства подробно описаны в главе 23.)

Не поступайте так в реальных проектах

Подход, принятый в листинге 8.2, должен использоваться только для генерации тестовых данных и ни в какой другой части приложения.

В этой главе нужен механизм, который позволяет надежно генерировать крупные объемы тестовых данных, не требуя выполнения сложных задач в БД или применения сторонних инструментов. (Для генерации данных SQL доступны великолепные коммерческие инструменты, но они обычно требуют приобретения лицензии на любой код, превышающий несколько сотен строк.)

При работе непосредственно с SQL следует соблюдать осторожность, поскольку в таком случае обходятся многочисленные полезные средства защиты, обеспечиваемые инфраструктурой Entity Framework Core. Итоговый код SQL трудно тестировать и сопровождать, к тому же зачастую оказывается, что он работает на единственном сервере баз данных. Кроме того, вы должны избегать создания хранимых процедур в коде C#, что для простоты делалось в листинге 8.2.

Короче говоря, не используйте никакие аспекты продемонстрированного приема в производственных частях своих приложений.

Чтобы снабдить контроллер представлением, создайте папку `Views/Seed` и поместите в нее файл по имени `Index.cshtml` с содержимым из листинга 8.3.

Листинг 8.3. Содержимое файла `Index.cshtml` из папки `Views/Seed`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Seed Data</h3>

<form method="post">
  <div class="form-group">
    <label>Number of Objects to Create</label>
    <input class="form-control" name="count" value="50" />
  </div>
  <div class="text-center">
    <button type="submit" asp-action="CreateSeedData"
            class="btn btn-primary">
      Seed Database
    </button>
    <button asp-action="ClearData" class="btn btn-danger">
      Clear Database
    </button>
  </div>
</form>

<h5 class="text-center m-2">
    There are @ViewBag.Count products in the database
</h5>

<div class="container-fluid">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase</div>
    <div class="col font-weight-bold text-right">Retail</div>
  </div>
  @foreach (Product p in Model) {
    <div class="row">
      <div class="col-1">@p.Id</div>
      <div class="col">@p.Name</div>
      <div class="col">@p.Category.Name</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
    </div>
  }
</div>
```

Представление позволяет указывать, сколько тестовых данных должно быть сгенерировано, и отображает первые 20 объектов `Product`, которые предоставляются запросом в действии `Index` контроллера `Seed`. Чтобы облегчить работу с контроллером `Seed`, добавьте в разделяемую компоновку элемент, показанный в листинге 8.4.

Листинг 8.4. Добавление элемента в файле `_Layout.cshtml` из папки `Views/Shared`

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

```
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <style>
    .placeholder { visibility: collapse; display: none }
    .placeholder:only-child { visibility: visible; display: flex }
  </style>
</head>
<body>
  <div class="container-fluid">
    <div class="row p-2">
      <div class="col-2">
        <a asp-controller="Home" asp-action="Index"
           class="@GetClassForButton("Home")">
          Products
        </a>
        <a asp-controller="Categories" asp-action="Index"
           class="@GetClassForButton("Categories")">
          Categories
        </a>
        <a asp-controller="Orders" asp-action="Index"
           class="@GetClassForButton("Orders")">
          Orders
        </a>
        <a asp-controller="Seed" asp-action="Index"
           class="@GetClassForButton("Seed")">
          Seed Data
        </a>
      </div>
      <div class="col">
        @RenderBody()
      </div>
    </div>
  </div>
</body>
</html>

@functions {
  string GetClassForButton(string controller) {
    return "btn btn-block " + (ViewContext.RouteData.Values["controller"]
      as string == controller ? "btn-primary" : "btn-outline-primary");
  }
}
```

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Seed Data (Начальные данные). Укажите значение `1000` в элементе `input` и щелкните на кнопке Seed Database (Заполнить БД начальными данными). Генерация данных займет какое-то время, после чего вы увидите результат, приведенный на рис. 8.1.

На заметку! Значения цен для тестовых данных генерируются случайным образом, а потому в некоторых примерах вы можете получить слегка отличающиеся результаты.

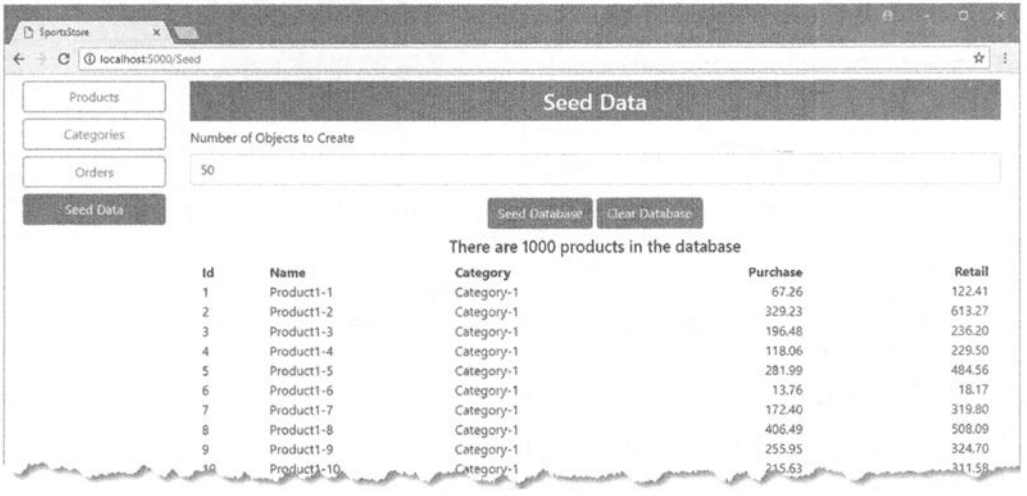

Рис. 8.1. Выполнение примера приложения

Масштабирование представления данных

Чтобы продемонстрировать изъяны способа, которым приложение `SportsStore` представляет свои данные, особо много данных не требуется. При наличии тысячи объектов метод представления данных пользователю становится непригодным, и это все еще относительно небольшой объем данных, с которым приложению приходится иметь дело. В последующих разделах способ представления данных приложением `SportsStore` будет изменен, чтобы помочь пользователю выполнять основные операции и находить интересующие его объекты.

Добавление поддержки разбиения на страницы

Первой будет решена задача разбиения данных, представляемых пользователю, с тем, чтобы они не выглядели как один длинный список. Использование простых таблиц, которые включают все объекты, является удобным подходом, когда формируются основы приложения, но таблицы, содержащие тысячи строк, непригодны для применения в большинстве приложений. Чтобы решить эту проблему, мы добавим поддержку запрашивания у БД небольших объемов данных и позволим пользователю перемещаться, перелистывая такие страницы с небольшими объемами данных.

При работе с крупными объемами данных важно обеспечить согласованное управление доступом к этим данным, чтобы у одной части приложения не было никакой возможности случайно запросить миллионы объектов. Мы примем подход, предусматривающий создание класса коллекции, который заключает в себе разбиение на страницы.

Создание класса коллекции с поддержкой разбиения на страницы

Для определения коллекции, которая обеспечит доступ к разбитым на страницы данным, создайте папку `Models/Pages`, добавьте в нее файл класса по имени `PagedList.cs` и поместите в него содержимое листинга 8.5.

Листинг 8.5. Содержимое файла PagedList.cs из папки Models/Pages

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

namespace SportsStore.Models.Pages {

  public class PagedList<T> : List<T> {

    public PagedList(IQueryable<T> query, QueryOptions options = null) {
      CurrentPage = options.CurrentPage;
      PageSize = options.PageSize;
      TotalPages = query.Count() / PageSize;
      AddRange(query.Skip((CurrentPage - 1) * PageSize).Take(PageSize));
    }

    public int CurrentPage { get; set; }
    public int PageSize { get; set; }
    public int TotalPages { get; set; }

    public bool HasPreviousPage => CurrentPage > 1;
    public bool HasNextPage => CurrentPage < TotalPages;
  }
}
```

В качестве базового класса используется строго типизированный `List`, который позволит легко наращивать базовое поведение коллекции. Конструктор принимает объект реализации `IQueryable<T>`, представляющий запрос, который будет снабжать данными для отображения пользователю. Такой запрос будет выполняться дважды — один раз для получения общего количества объектов, которое мог бы возвратить запрос, и один раз для получения только объектов, подлежащих отображению на текущей странице. Это компромисс, присущий разбиению на страницы, при котором дополнительный запрос `COUNT` уравновешивается запросами для меньшего количества объектов в целом. В остальных аргументах конструктора указываются страница, требуемая для запроса, и количество объектов, отображаемых на странице.

Для представления параметров, необходимых запросу, добавьте в папку `Models/Pages` файл класса по имени `QueryOptions.cs` с кодом, показанным в листинге 8.6.

Листинг 8.6. Содержимое файла QueryOptions.cs из папки Models/Pages

```
namespace SportsStore.Models.Pages {

  public class QueryOptions {
    public int CurrentPage { get; set; } = 1;
    public int PageSize { get; set; } = 10;
  }
}
```

Обновление хранилища

Чтобы обеспечить согласованное применение средства разбиения на страницы, в качестве результата выполняемых в хранилище запросов будет возвращаться объект `PagedList`. Добавьте в интерфейс `IRepository` метод по имени `GetProduct()`, который возвращает данные для одиночной страницы (листинг 8.7).

Листинг 8.7. Возвращение страницы данных в файле IRepository.cs из папки Models

```
using System.Collections.Generic;
using SportsStore.Models.Pages;

namespace SportsStore.Models {

  public interface IRepository {

    IEnumerable<Product> Products { get; }

    PagedList<Product> GetProducts(QueryOptions options);

    Product GetProduct(long key);

    void AddProduct(Product product);

    void UpdateProduct(Product product);

    void UpdateAll(Product[] products);

    void Delete(Product product);
  }
}
```

Внесите соответствующие изменения в класс реализации хранилища (листинг 8.8).

Листинг 8.8. Возвращение страницы данных в файле DataRepository.cs из папки Models

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models.Pages;

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;

    public DataRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Product> Products => context.Products
      .Include(p => p.Category).ToArray();

    public PagedList<Product> GetProducts(QueryOptions options) {
      return new PagedList<Product>(context.Products
        .Include(p => p.Category), options);
    }

    // ...для краткости остальные методы не показаны...

  }
}
```

Новый метод возвращает коллекцию PagedList объектов Product для страницы, указанной аргументами.

Обновление контроллера и представления

Для добавления в контроллер Home поддержки разбиения на страницы измените действие Index, чтобы оно принимало аргументы, необходимые при выборе страницы, и в результате использовало новый метод хранилища (листинг 8.9).

Листинг 8.9. Применение данных, разбитых на страницы, в файле HomeController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class HomeController : Controller {
    private IRepository repository;
    private ICategoryRepository catRepository;

    public HomeController(IRepository repo, ICategoryRepository catRepo)
{
      repository = repo;
      catRepository = catRepo;
    }

    public IActionResult Index(QueryOptions options) {
      return View(repository.GetProducts(options));
    }

    public IActionResult UpdateProduct(long key) {
      ViewBag.Categories = catRepository.Categories;
      return View(key == 0 ? new Product() : repository.GetProduct(key));
    }

    [HttpPost]
    public IActionResult UpdateProduct(Product product) {
      if (product.Id == 0) {
        repository.AddProduct(product);
      } else {
        repository.UpdateProduct(product);
      }
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Delete(Product product) {
      repository.Delete(product);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Базовый класс коллекции данных, разбитых на страницы, реализует интерфейс `IEnumerable<T>`, который сводит к минимуму объем изменений, требующихся для поддержки разбитых на страницы данных. Единственное изменение, которое понадобится внести в представление для действия `Index` контроллера `Home`, связано с отображением частичного представления с деталями разбиения на страницы (листинг 8.10). Остаток представления в изменении не нуждается, поскольку данные будут перечисляться тем же способом вне зависимости от того, содержит перечисляемая последовательность все доступные данные или только страницу данных.

Листинг 8.10. Использование частичного представления в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="text-center">
  @Html.Partial("Pages", Model)
</div>

<div class="container-fluid mt-3">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase Price</div>
    <div class="col font-weight-bold text-right">Retail Price</div>
    <div class="col"></div>
  </div>
  @foreach (Product p in Model) {
    <div class="row p-2">
      <div class="col-1">@p.Id</div>
      <div class="col">@p.Name</div>
      <div class="col">@p.Category.Name</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
      <div class="col">
        <form asp-action="Delete" method="post">
          <a asp-action="UpdateProduct" asp-route-key="@p.Id"
             class="btn btn-outline-primary">
            Edit
          </a>
          <input type="hidden" name="Id" value="@p.Id" />
          <button type="submit" class="btn btn-outline-danger">
            Delete
          </button>
        </form>
      </div>
    </div>
  }
  <div class="text-center p-2">
    <a asp-action="UpdateProduct" asp-route-key="0"
       class="btn btn-primary">Add</a>
  </div>
</div>
```

Чтобы завершить поддержку разбиения на страницы для объектов `Product`, определите частичное представление, добавив в папку `Views/Shared` файл по имени `Pages.cshtml` с элементами, которые показаны в листинге 8.11.

Листинг 8.11. Содержимое файла `Pages.cshtml` из папки `Views/Shared`

```
<form id="pageform" method="get" class="form-inline d-inline-block">
  <button name="options.currentPage" value="@(Model.CurrentPage -1)"
```

```
        class=
"btn btn-outline-primary @(!Model.HasPreviousPage ? "disabled" : "")"
        type="submit">
    Previous
  </button>
  @for (int i = 1; i <= 3 && i <= Model.TotalPages; i++) {
    <button name="options.currentPage" value="@i" type="submit"
          class="btn btn-outline-primary
                @(Model.CurrentPage == i ? "active" : "")">
      @i
    </button>
  }
  @if (Model.CurrentPage > 3 && Model.TotalPages - Model.CurrentPage >= 3) {
    @:...
    <button class="btn btn-outline-primary active">@Model.CurrentPage
    </button>
  }
  @if (Model.TotalPages > 3) {
    @:...
    @for (int i = Math.Max(4, Model.TotalPages - 2);
            i <= Model.TotalPages; i++) {
      <button name="options.currentPage" value="@i" type="submit"
            class="btn btn-outline-primary
                  @(Model.CurrentPage == i ? "active" : "")">
        @i
      </button>
    }
  }
  <button name="options.currentPage" value="@(Model.CurrentPage +1)"
          type="submit"
          class="btn btn-outline-primary
                @(!Model.HasNextPage ? "disabled" : "")">
    Next
  </button>
  <select name="options.pageSize" class="form-control ml-1 mr-1">
    @foreach (int val in new int[] { 10, 25, 50, 100 }) {
      <option value="@val" selected="@(Model.PageSize == val)">@val
      </option>
    }
  </select>
  <input type="hidden" name="options.currentPage" value="1" />
  <button type="submit" class="btn btn-secondary">Change Page Size
  </button>
</form>
```

Представление содержит HTML-форму, которая применяется для отправки HTTP-запросов `GET` обратно методу действия страниц данных и для изменения размера страницы. Выражения Razor выглядят запутанными, но они приспосабливают кнопки страниц, отображаемые пользователю, к имеющемуся количеству страниц. Чтобы увидеть эффект, запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`. Список товаров будет разбит на страницы по 10 элементов, на которые можно переходить с помощью последовательности кнопок (рис. 8.2).

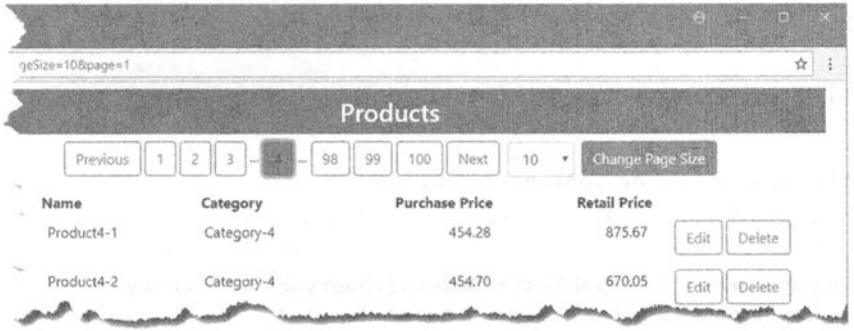

Рис. 8.2. Разбиение данных на страницы

Добавление поддержки поиска и упорядочения

Отображение страниц — хорошее начало, но концентрироваться на специфическом наборе объектов по-прежнему трудно. Для оснащения пользователя инструментами нахождения интересующих данных мы добавим к разбиению на страницы поддержку изменения порядка отображения и выполнения поиска. Отправной точкой будет расширение класса `PagedList`, чтобы он мог выполнять поиск и упорядочивать результаты запросов с использованием имен свойств, а не лямбда-выражений, выбирающих свойства (листинг 8.12). Выполнение таких операций потребует несколько запутанного кода, но результат может быть применен к любому классу модели данных и легче интегрируется с частями ASP.NET Core MVC приложения.

Листинг 8.12. Добавление функциональных средств в файле `PagedList.cs` из папки `Models/Pages`

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

namespace SportsStore.Models.Pages {

  public class PagedList<T> : List<T> {

    public PagedList(IQueryable<T> query, QueryOptions options = null) {
      CurrentPage = options.CurrentPage;
      PageSize = options.PageSize;
      Options = options;

      if (options != null) {
        if (!string.IsNullOrEmpty(options.OrderPropertyName)) {
          query = Order(query, options.OrderPropertyName,
            options.DescendingOrder);
        }
        if (!string.IsNullOrEmpty(options.SearchPropertyName)
            && !string.IsNullOrEmpty(options.SearchTerm)) {
          query = Search(query, options.SearchPropertyName,
            options.SearchTerm);
        }
      }
```

```
        TotalPages = query.Count() / PageSize;
        AddRange(query.Skip((CurrentPage - 1) * PageSize).Take(PageSize));
    }
    public int CurrentPage { get; set; }
    public int PageSize { get; set; }
    public int TotalPages { get; set; }
    public QueryOptions Options { get; set; }

    public bool HasPreviousPage => CurrentPage > 1;
    public bool HasNextPage => CurrentPage < TotalPages;

    private static IQueryable<T> Search(IQueryable<T> query,
        string propertyName, string searchTerm) {
      var parameter = Expression.Parameter(typeof(T), "x");
      var source = propertyName.Split('.').Aggregate((Expression) parameter,
        Expression.Property);
      var body = Expression.Call(source, "Contains", Type.EmptyTypes,
        Expression.Constant(searchTerm, typeof(string)));
      var lambda = Expression.Lambda<Func<T, bool>>(body, parameter);
      return query.Where(lambda);
    }
    private static IQueryable<T> Order(IQueryable<T> query,
        string propertyName, bool desc) {
      var parameter = Expression.Parameter(typeof(T), "x");
      var source = propertyName.Split('.').Aggregate((Expression) parameter,
        Expression.Property);
      var lambda = Expression.Lambda(typeof(Func<,>).MakeGenericType(typeof(T),
        source.Type), source, parameter);
      return typeof(Queryable).GetMethods().Single(
          method => method.Name == (desc ? "OrderByDescending"
                : "OrderBy")
          && method.IsGenericMethodDefinition
          && method.GetGenericArguments().Length == 2
          && method.GetParameters().Length == 2)
        .MakeGenericMethod(typeof(T), source.Type)
        .Invoke(null, new object[] { query, lambda }) as IQueryable<T>;
    }
  }
}
```

В листинге 8.13 приведены соответствующие изменения в классе `QueryOptions`.

Листинг 8.13. Добавление свойств в файле `QueryOptions.cs` из папки `Models/Pages`

```
namespace SportsStore.Models.Pages {

  public class QueryOptions {

    public int CurrentPage { get; set; } = 1;
    public int PageSize { get; set; } = 10;

    public string OrderPropertyName { get; set; }
    public bool DescendingOrder { get; set; }

    public string SearchPropertyName { get; set; }
    public string SearchTerm { get; set; }
  }
}
```

Чтобы создать обобщенное представление, которое будет предлагать пользователю возможности поиска и упорядочения, добавьте в папку `Views/Shared` файл по имени `PageOptions.cshtml` с содержимым из листинга 8.14.

Листинг 8.14. Содержимое файла `PageOptions.cshtml` из папки `Views/Shared`

```
<div class="container-fluid mt-2">
  <div class="row m-1">
    <div class="col"></div>
    <div class="col-1">
      <label class="col-form-label">Search</label>
    </div>
    <div class="col-3">
      <select form="pageform" name="options.searchpropertyname"
            class="form-control">
        @foreach (string s in ViewBag.searches as string[]) {
          <option value="@s"
                selected="@(Model.Options.SearchPropertyName == s)">
            @(s.IndexOf('.') == -1 ? s : s.Substring(0, s.IndexOf('.')))
          </option>
        }
      </select>
    </div>
    <div class="col">
      <input form="pageform" class="form-control"
             name="options.searchterm"
            value="@Model.Options.SearchTerm" />
    </div>
    <div class="col-1 text-right">
      <button form="pageform" class="btn btn-secondary" type="submit">
        Search
      </button>
    </div>
    <div class="col"></div>
  </div>
  <div class="row m-1">
    <div class="col"></div>
    <div class="col-1">
      <label class="col-form-label">Sort</label>
    </div>
    <div class="col-3">
      <select form="pageform" name="options.OrderPropertyName"
            class="form-control">
        @foreach (string s in ViewBag.sorts as string[]) {
          <option value="@s"
                selected="@(Model.Options.OrderPropertyName == s)">
            @(s.IndexOf('.') == -1 ? s : s.Substring(0, s.IndexOf('.')))
          </option>
        }
      </select>
    </div>
    <div class="col form-check form-check-inline">
      <input form="pageform" type="checkbox" name="Options.DescendingOrder"
```

```
            id="Options.DescendingOrder"
            class="form-check-input" value="true"
            checked ="@Model.Options.DescendingOrder" />
      <label class="form-check-label">Descending Order</label>
    </div>
    <div class="col-1 text-right">
      <button form="pageform" class="btn btn-secondary" type="submit">
        Sort
      </button>
    </div>
    <div class="col"></div>
  </div>
</div>
```

Представление опирается на возможность HTML 5 ассоциировать с формами элементы, определенные за пределами элемента `form`. Это позволяет расширить форму, которая определена в представлении `Pages`, элементами, специфичными для поиска и упорядочения.

Жестко кодировать список свойств, которые пользователь может задействовать для поиска или упорядочения данных в представлении, нежелательно, поэтому в целях простоты такие значения получаются из `ViewBag`. Решение не считается идеальным, но оно обеспечивает высокую гибкость и позволяет легко адаптировать одно и то же содержимое к разным представлениям и данным. Чтобы отобразить пользователю элементы, связанные с поиском и упорядочением, рядом со списком объектов `Product`, добавьте в представление `Index`, используемое контроллером `Home`, содержимое из листинга 8.15.

Листинг 8.15. Отображение элементов, связанных с поиском и упорядочением, в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Products</h3>

<div class="text-center">
  @Html.Partial("Pages", Model)
  @{
    ViewBag.searches = new string[] { "Name", "Category.Name" };
    ViewBag.sorts = new string[] { "Name", "Category.Name",
      "PurchasePrice", "RetailPrice"};
  }
  @Html.Partial("PageOptions", Model)
</div>
<div class="container-fluid mt-3">
  <!-- ...для краткости остальные элементы не показаны... -->
</div>
```

Блок кода указывает свойства класса `Product`, с помощью которых пользователь получит возможность искать и упорядочивать объекты `Product`, а выражение `@Html.Partial` визуализирует элементы для таких средств.

Для просмотра результатов запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`. Вы увидите новую последовательность элементов, облегчающих навигацию по данным (рис. 8.3).

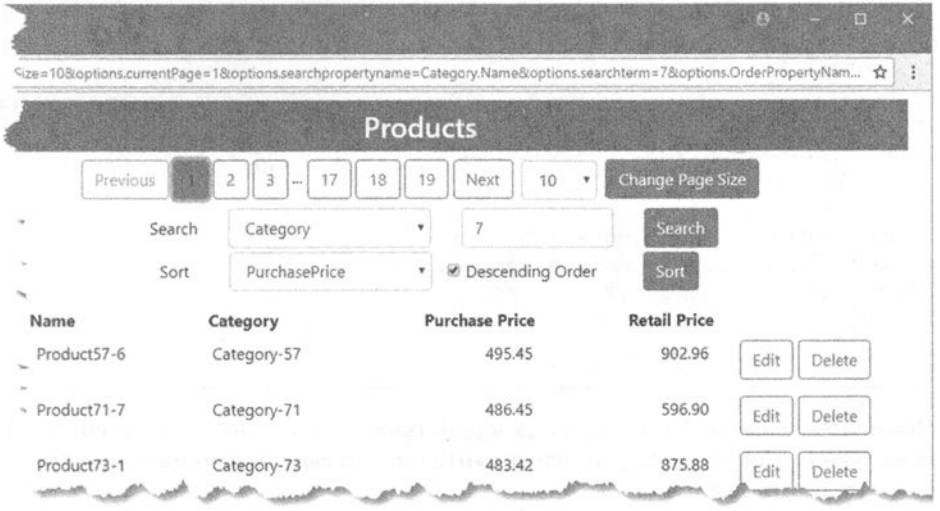

Рис. 8.3. Поиск и упорядочение объектов `Product`

Применение возможностей представления данных к категориям

Процесс размещения средств для разбиения на страницы, поиска и сортировки по своим местам был непростым, но теперь, когда фундамент готов, их можно применить к другим типам данных в приложении, таким как объекты `Category`. Первым делом обновите интерфейс и классы реализации, добавив метод, который принимает объект `QueryOptions` и возвращает результат `PagedList` (листинг 8.16).

Листинг 8.16. Добавление поддержки страниц в файле `CategoryRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using SportsStore.Models.Pages;

namespace SportsStore.Models {

  public interface ICategoryRepository {

    IEnumerable<Category> Categories { get; }

    PagedList<Category> GetCategories(QueryOptions options);

    void AddCategory(Category category);
    void UpdateCategory(Category category);
    void DeleteCategory(Category category);
  }

  public class CategoryRepository : ICategoryRepository {
    private DataContext context;

    public CategoryRepository(DataContext ctx) => context = ctx;

    public IEnumerable<Category> Categories => context.Categories;

    public PagedList<Category> GetCategories(QueryOptions options) {
      return new PagedList<Category>(context.Categories, options);
    }
```

```
    public void AddCategory(Category category) {
      context.Categories.Add(category);
      context.SaveChanges();
    }
    public void UpdateCategory(Category category) {
      context.Categories.Update(category);
      context.SaveChanges();
    }
    public void DeleteCategory(Category category) {
      context.Categories.Remove(category);
      context.SaveChanges();
    }
  }
}
```

Добавьте параметр `QueryOptions` к действию `Index` контроллера, который управляет объектами `Category`, и используйте его для запрашивания хранилища, как показано в листинге 8.17.

Листинг 8.17. Добавление поддержки страниц в файле `CategoriesController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
  public class CategoriesController : Controller {
    private ICategoryRepository repository;

    public CategoriesController(ICategoryRepository repo)
     => repository = repo;
    public IActionResult Index(QueryOptions options)
      => View(repository.GetCategories(options));
    [HttpPost]
    public IActionResult AddCategory(Category category) {
      repository.AddCategory(category);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult EditCategory(long id) {
      ViewBag.EditId = id;
      return View("Index", repository.Categories);
    }
    [HttpPost]
    public IActionResult UpdateCategory(Category category) {
      repository.UpdateCategory(category);
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult DeleteCategory(Category category) {
      repository.DeleteCategory(category);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Наконец, предоставьте пользователю возможность работы со средствами, добавив элементы из листинга 8.18 в представление `Index`, которое применяется контроллером `Categories`.

Листинг 8.18. Добавление возможностей в файле `Index.cshtml` из папки `Views/Categories`

```
@model IEnumerable<Category>
<h3 class="p-2 bg-primary text-white text-center">Categories</h3>
<div class="text-center">
  @Html.Partial("Pages", Model)
  @{
    ViewBag.searches = new string[] { "Name", "Description" };
    ViewBag.sorts = new string[] { "Name", "Description" };
  }
  @Html.Partial("PageOptions", Model)
</div>
<div class="container-fluid mt-3">
  <!-- ...для краткости остальные элементы не показаны... -->
</div>
```

Чтобы увидеть новые средства, запустите приложение, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Categories (Категории). На страницах присутствует список категорий, а пользователь может выполнять в них поиск и упорядочение (рис. 8.4).

Рис. 8.4. Разбиение на страницы, упорядочение и поиск категорий

Индексация базы данных

Добавления в БД тысячи тестовых объектов оказалось достаточно для демонстрации ограничений способа, которым данные представлялись пользователю, но такого объема данных не хватит, чтобы выявить ограничения самой БД. Для выяснения эффекта от работы с крупными объемами данных добавьте к конструктору `PagedList` операторы, которые измеряют длительность выполнения запроса и выводят затраченное время на консоль (листинг 8.19).

Совет. Существует много способов измерения производительности, и большинство серверов баз данных предлагают инструменты, которые помогут понять, сколько времени потребуется запросам на выполнение. В случае SQL Server инструменты SQL Server Profiler и SQL Server Management Studio предоставляют буквально бесчисленные детали. Хотя упомянутые инструменты могут быть удобными, обычно я полагаюсь на подход, принятый в листинге 8.19, т.к. он простой и достаточно точный, чтобы понять значимость проблем с производительностью.

Листинг 8.19. Определение времени выполнения запроса в файле `PagedList.cs` из папки `Models/Pages`

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

namespace SportsStore.Models.Pages {

  public class PagedList<T> : List<T> {

    public PagedList(IQueryable<T> query, QueryOptions options = null) {
      CurrentPage = options.CurrentPage;
      PageSize = options.PageSize;
      Options = options;

      if (options != null) {
        if (!string.IsNullOrEmpty(options.OrderPropertyName)) {
          query = Order(query, options.OrderPropertyName,
            options.DescendingOrder);
        }
        if (!string.IsNullOrEmpty(options.SearchPropertyName)
            && !string.IsNullOrEmpty(options.SearchTerm)) {
          query = Search(query, options.SearchPropertyName,
            options.SearchTerm);
        }
      }

      Stopwatch sw = Stopwatch.StartNew();
      Console.Clear();

      TotalPages = query.Count() / PageSize;
      AddRange(query.Skip((CurrentPage - 1) * PageSize).Take(PageSize));

      Console.WriteLine($"Query Time: {sw.ElapsedMilliseconds} ms");
    }
    // ...для краткости остальные члены не показаны...
  }
}
```

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Seed Data и заполните БД нужным количеством тестовых объектов. После того, как БД заполнится начальными данными, щелкните на кнопке Products (Товары), выберите в списке Sort (Сортировка по) свойство Purchase Price (Покупная цена), отметьте флажок Descending Order (Убывающий порядок) и щелкните на кнопке Sort (Сортировать).

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то заметите запросы, которые применялись для получения данных, а также время их выполнения:

```
...
SELECT COUNT(*)
FROM [Products] AS [p]
...
SELECT [p].[Id], [p].[CategoryId], [p].[Name], [p].[PurchasePrice],
  [p].[RetailPrice], [p.Category].[Id], [p.Category].[Description],
  [p.Category].[Name
FROM [Products] AS [p]
INNER JOIN [Categories] AS [p.Category] ON [p].[CategoryId] =
[p.Category].[Id]
ORDER BY [p].[PurchasePrice] DESC
OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY
...
Query Time: 14 ms
Время выполнения запроса: 14 мс
...
```

В табл. 8.1 приведены показатели времени, которое заняло выполнение этих запросов на моей машине разработки, для разных объемов начальных данных. Вы можете получить другие показатели, но важно отметить, что с ростом объема данных они увеличиваются.

Таблица 8.1. Время, которое заняло выполнение запросов

Количество объектов	Время
1000	14 мс
10 000	17 мс
100 000	185 мс
1 000 000	2453 мс
2 000 000	5713 мс

Создание и применение индексов

Одна из проблем с производительностью возникает из-за того, что серверу баз данных приходится исследовать много строк данных в поиске данных, необходимых приложению. Эффективный способ сокращения объема работы, выполняемой сервером баз данных, предусматривает создание индексов, которые ускоряют запросы, но требуют определенного времени на первоначальную подготовку и небольшой дополнительной работы после каждого обновления. Что касается приложения `SportsStore`, то мы добавим индексы для свойств классов `Product` и `Category`, которые пользователь сможет использовать при поиске или упорядочении данных. Создайте индексы в классе контекста БД, как показано в листинге 8.20.

Листинг 8.20. Создание индексов в файле `DataContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {

  public class DataContext : DbContext {

    public DataContext(DbContextOptions<DataContext> opts) : base(opts) {}

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

    public DbSet<Category> Categories { get; set; }

    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderLine> OrderLines { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Product>().HasIndex(p => p.Name);
      modelBuilder.Entity<Product>().HasIndex(p => p.PurchasePrice);
      modelBuilder.Entity<Product>().HasIndex(p => p.RetailPrice);
      modelBuilder.Entity<Category>().HasIndex(p => p.Name);
      modelBuilder.Entity<Category>().HasIndex(p => p.Description);
    }
  }
}
```

Метод `OnModelCreating()` переопределен для настройки модели данных с применением средства Fluent API из Entity Framework Core, которое будет подробно описано в частях II и III книги. Интерфейс Fluent API позволяет переопределять стандартные линии поведения Entity Framework Core и получать доступ к расширенным возможностям, таким как создание индексов. В листинге 8.20 создаются индексы для свойств `Name`, `PurchasePrice` и `RetailPrice` класса `Product`, а также для свойств `Name` и `Description` класса `Category`. Создавать индексы для свойств первичных или внешних ключей не нужно, поскольку инфраструктура Entity Framework Core по умолчанию создает их самостоятельно.

Создание индексов требует создания новой миграции и ее применения к БД. Выполните команды, приведенные в листинге 8.21, в папке проекта `SportsStore`, чтобы создать миграцию по имени `Indexes` и применить ее к БД.

Совет. При наличии в БД большого объема данных применение миграции, создающей индексы, может занять некоторое время, т.к. все существующие данные должны быть добавлены в индекс. Перед выполнением команд миграции возможно у вас возникнет желание воспользоваться контроллером `Seed` для сокращения объема тестовых данных.

Листинг 8.21. Создание и применение миграции БД

```
dotnet ef migrations add Indexes
dotnet ef database update
```

После того, как миграция применена, перезапустите приложение и повторите выполненные ранее тестовые запросы, чтобы посмотреть, как индексы повлияли на производительность. В табл. 8.2 представлены показатели времени, которое потребовалось на выполнение тестовых запросов на моей машине разработки до и после добавления индексов.

Таблица 8.2. Время выполнения запросов

Количество объектов	Время без индексов	Время с индексами
1000	14 мс	9 мс
10 000	17 мс	10 мс
100 000	185 мс	23 мс
1 000 000	2453 мс	143 мс
2 000 000	5713 мс	158 мс

Производительность запроса COUNT

По мере роста объема данных по-прежнему наблюдается небольшое увеличение времени выполнения запросов. Запрос, предназначенный для получения количества хранящихся в БД объектов, транслируется в SQL-команду `SELECT COUNT`, производительность которой падает для крупного количества объектов. Серверы баз данных обычно предлагают альтернативные методы подсчета данных, и в случае SQL Server можно запрашивать метаданные, которые сервер поддерживает для БД:

```
...
select sum (spart.rows)
from sys.partitions spart
where spart.object_id = object_id('Products') and spart.index_id < 2
...
```

Запрос такого типа не может быть выполнен с использованием LINQ. Обратитесь в главу 23, где показано, как применять средства Entity Framework Core, которые позволяют выполнять SQL-команды напрямую.

Распространенные проблемы и их решения

Масштабирование объема данных, который поддерживается приложением, требует тщательной подгонки части ASP.NET Core MVC приложения для запрашивания у инфраструктуры Entity Framework Core меньших объемов данных и предоставления инструментов для упорядочения и поиска данных. В последующих разделах будут описаны проблемы, с которыми вы вероятнее всего столкнетесь, и приведены объяснения, как их решить.

Запросы для страниц выполняются слишком медленно

Наиболее вероятная причина медленных запросов в том, что приложение извлекает все объекты из БД и затем сортирует или производит поиск в памяти, прежде чем взять только те объекты, которые требуются для отдельной страницы. Каждый раз, когда пользователь переходит на новую страницу, этот процесс повторяется, порождая большие объемы затрат по извлечению и обработке объектов, которые впоследствии просто отбрасываются.

Проблема обычно возникает из-за вызова методов LINQ на объекте реализации интерфейса `IEnumerable<T>`, а не интерфейса `IQueryable<T>`, как было описано

в главе 5. Установить наличие проблемы быстрее всего, просмотрев в журнальных сообщениях приложения SQL-запросы, которые генерирует инфраструктура Entity Framework Core. Хотя детали могут варьироваться, использование LINQ-методов `OrderBy()` и `Skip()` с объектом реализации `IQueryable<T>` будет производить запросы с конструкциями `ORDER BY` и `OFFSET`.

В случае применения интерфейса `IQueryable<T>` вы должны выполнить проверку на предмет дублированных запросов, как объяснялось в главе 5. Довольно легко забыть о том, что перечисление последовательности объектов будет инициировать запрос, особенно при выяснении требуемого количества кнопок для перехода на страницы, например.

Во время применения миграции, добавляющей индексы, возникает тайм-аут

Когда к БД применяется миграция, добавляющая индексы, сервер баз данных должен заполнить индексы с использованием хранящихся данных. Для крупной БД процесс бывает длительным, и у команды `dotnet ef` может возникнуть тайм-аут до того, как процесс завершится, что приведет к отказу миграции и отсутствию индексов. Чтобы решить проблему во время разработки, удалите и воссоздайте БД, чтобы индексы создавались для пустой БД. В производственной системе выполните резервное копирование БД, удалите данные и затем примените миграции. После того, как индексы созданы, можете заполнить БД снова, используя небольшие блоки данных, чтобы каждое обновление требовало меньшего объема работы.

Создание индекса не улучшило производительность

Если вы обнаруживаете, что индекс не улучшает производительность запросов, тогда первым делом проверьте, применялась ли к БД миграция, которая создает индекс. Еще одна вероятная причина заключается в том, что индексы не были созданы для всех свойств, которые приложение использует в запросах. Если ваше приложение при поиске применяет сочетания свойств, то может понадобиться создать дополнительные индексы, как обсуждается в части III книги.

Резюме

В главе было показано, каким образом адаптировать приложение `SportsStore` для работы с более крупными объемами данных. Мы добавили к нему поддержку разбиения на страницы, упорядочения и поиска данных, что позволяет пользователю иметь дело с управляемым количеством объектов за раз. Кроме того, с помощью Fluent API была настроена модель данных и добавлены индексы для улучшения производительности запросов. В следующей главе к приложению `SportsStore` будет добавлен интерфейс для покупателей.

ГЛАВА 9

SportsStore: интерфейс для покупателей

В этой главе будут построены части приложения SportsStore, реализующие интерфейс для покупателей, который позволит выбирать товары, просматривать корзину для покупок и оформлять заказ. Добавляемые функциональные средства в значительной степени связаны с инфраструктурой ASP.NET Core MVC и основаны на фундаменте Entity Framework Core, который был создан в предшествующих главах.

Изложение материала в главе будет вестись ускоренными темпами, поскольку большая часть работы касается построения средств с использованием ASP.NET Core MVC поверх фундамента, сформированного с помощью инфраструктуры Entity Framework Core в предыдущих главах.

Подготовительные шаги

Мы продолжим использовать проект SportsStore, который был создан в главе 4 и модифицирован в последующих главах.

Совет. Проект SportsStore и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Удаление операторов, которые измеряют длительность выполнения запросов

В главе 8 измерялось время, в течение которого выполнялись запросы. Операторы измерения времени больше не нужны, поэтому закомментируйте их, как показано в листинге 9.1.

Листинг 9.1. Комментирование операторов в файле PagedList.cs из папки Models/Pages

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

```
using System.Linq.Expressions;
using System.Diagnostics;

namespace SportsStore.Models.Pages {

  public class PagedList<T> : List<T> {

    public PagedList(IQueryable<T> query, QueryOptions options = null) {
      CurrentPage = options.CurrentPage;
      PageSize = options.PageSize;
      Options = options;
      if (options != null) {
        if (!string.IsNullOrEmpty(options.OrderPropertyName)) {
          query = Order(query, options.OrderPropertyName,
          options.DescendingOrder);
        }
        if (!string.IsNullOrEmpty(options.SearchPropertyName)
            && !string.IsNullOrEmpty(options.SearchTerm)) {
          query = Search(query, options.SearchPropertyName,
            options.SearchTerm);
        }
      }
      // Stopwatch sw = Stopwatch.StartNew();
      // Console.Clear();
      TotalPages = query.Count() / PageSize;
      AddRange(query.Skip((CurrentPage - 1) * PageSize).Take(PageSize));
      // Console.WriteLine($"Query Time: {sw.ElapsedMilliseconds} ms");
    }

    // ...для краткости остальные члены не показаны...
  }
}
```

Добавление импорта представления

В главе 8 класс `PagedList` применялся в представлениях без модификации модели представления просто для демонстрации того, что средства масштабирования можно добавить, внеся минимальные изменения. В настоящей главе класс `PagedList` будет использоваться в представлениях напрямую, поэтому добавьте содержащее его пространство имен в файл импортирования представлений (листинг 9.2).

Листинг 9.2. Добавление пространства имен в файле `_ViewImports.cshtml` из папки `Views`

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

Модификация модели данных

Чтобы подготовить модель данных к работе со средствами интерфейса для покупателей, добавьте в класс `Product` свойство `Description`, которое позволит покупателям получить немного сведений о просматриваемых товарах (листинг 9.3).

Листинг 9.3. Добавление свойства в файле `Products.cs` из папки `Models`

```
namespace SportsStore.Models {
  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal PurchasePrice { get; set; }
    public decimal RetailPrice { get; set; }
    public long CategoryId { get; set; }
    public Category Category { get; set; }
  }
}
```

Для облегчения запроса и сохранения данных по категории добавьте в класс `Category` навигационное свойство, которое инфраструктура Entity Framework Core сможет заполнять связанными объектами `Product` (листинг 9.4).

Листинг 9.4. Добавление навигационного свойства в файле `Category.cs` из папки `Models`

```
using System.Collections.Generic;
namespace SportsStore.Models {
  public class Category {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public IEnumerable<Product> Products { get; set; }
  }
}
```

Добавление начальных данных о товарах

Нам необходима возможность переключения между крупными объемами тестовых данных и небольшими объемами реалистичных данных. С этой целью добавьте в контроллер `Seed` код из листинга 9.5, который обеспечит наличие стандартных категорий и товаров `SportsStore`.

Листинг 9.5. Добавление начальных данных о товарах в файле `SeedController.cs` из папки `Controllers`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;
using System.Linq;
namespace SportsStore.Controllers {
  public class SeedController : Controller {
    private DataContext context;
    public SeedController(DataContext ctx) => context = ctx;
```

```
public IActionResult Index() {
  ViewBag.Count = context.Products.Count();
  return View(context.Products
    .Include(p => p.Category).OrderBy(p => p.Id).Take(20));
}

// ...для краткости остальные действия не показаны...

[HttpPost]
public IActionResult CreateProductionData() {
  ClearData();

  context.Categories.AddRange(new Category[] {
    new Category {
      Name = "Watersports",
           // Водный спорт
      Description = "Make a splash",
                  // Наделайте шуму
      Products = new Product[] {
        new Product {
          Name = "Kayak",
               // Каяк
          Description = "A boat for one person",
                      // Одноместная лодка
          PurchasePrice = 200, RetailPrice = 275
        },
        new Product {
          Name = "Lifejacket",
              // Спасательный жилет
          Description = "Protective and fashionable",
                      // Защитный и модный
          PurchasePrice = 40, RetailPrice = 48.95m
        },
      }
    },
    new Category {
      Name = "Soccer",
           // Футбол
      Description = "The World's Favorite Game",
                  // Любимая игра в мире
      Products = new Product[] {
        new Product {
          Name = "Soccer Ball",
               // Футбольный мяч
          Description = "FIFA-approved size and weight",
                      // Одобренный ФИФА размер и вес
          PurchasePrice = 18, RetailPrice = 19.50m
        },
        new Product {
          Name = "Corner Flags",
               // Угловые флажки
          Description = "Give your playing field a professional touch",
                      // Сделает ваше игровое поле профессиональным
          PurchasePrice = 32.50m, RetailPrice = 34.95m
        },
```

```
            new Product {
              Name = "Stadium",
                    // Стадион
              Description = "Flat-packed 35,000-seat stadium",
                           // Компактно упакованный стадион на 35 000 мест
              PurchasePrice = 75000, RetailPrice = 79500
            }
          }
        },
        new Category {
          Name = "Chess",
               // Шахматы
          Description = "The Thinky Game",
                       // Умственная игра
          Products = new Product[] {
            new Product {
              Name = "Thinking Cap",
                    // Мыслящая шапка
              Description = "Improve brain efficiency by 75%",
                           // Усиливает умственные способности на 75%
              PurchasePrice = 10, RetailPrice = 16
            },
            new Product {
              Name = "Unsteady Chair",
                    // Неустойчивый стул
              Description = "Secretly give your opponent a disadvantage",
                           // Незаметно создает сопернику неудобства
              PurchasePrice = 28, RetailPrice = 29.95m
            },
            new Product {
              Name = "Human Chess Board",
                     // Шахматная доска
              Description = "A fun game for the family",
                           // Веселая игра для всей семьи
              PurchasePrice = 68.50m, RetailPrice = 75
            },
            new Product {
              Name = "Bling-Bling King",
                    // Блестящий король
              Description = "Gold-plated, diamond-studded King",
                           // Позолоченный и усыпанный бриллиантами король
              PurchasePrice = 800, RetailPrice = 1200
            }
          }
        }
      });
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Новый метод действия создает последовательность объектов `Category` и устанавливает навигационное свойство `Products` в коллекцию объектов `Product`. Все объекты передаются методу `AddRange()` и сохраняются в БД методом `SaveChanges()`. Чтобы нацелиться на новый метод действия, добавьте в представление `Index`, применяемое контроллером `Seed`, элемент, выделенный полужирным в листинге 9.6.

Листинг 9.6. Добавление элемента в файле `Index.cshtml` из папки `Views/Seed`

```
@model IEnumerable<Product>

<h3 class="p-2 bg-primary text-white text-center">Seed Data</h3>

<form method="post">
  <div class="form-group">
    <label>Number of Objects to Create</label>
    <input class="form-control" name="count" value="50" />
  </div>
  <div class="text-center">
    <button type="submit" asp-action="CreateProductionData"
            class="btn btn-outline-primary">
      Production Seed
    </button>
    <button type="submit" asp-action="CreateSeedData"
            class="btn btn-primary">
      Seed Database
    </button>
    <button asp-action="ClearData" class="btn btn-danger">
      Clear Database
    </button>
  </div>
</form>

<h5 class="text-center m-2">
  There are @ViewBag.Count products in the database
</h5>

<div class="container-fluid">
  <div class="row">
    <div class="col-1 font-weight-bold">Id</div>
    <div class="col font-weight-bold">Name</div>
    <div class="col font-weight-bold">Category</div>
    <div class="col font-weight-bold text-right">Purchase</div>
    <div class="col font-weight-bold text-right">Retail</div>
  </div>
  @foreach (Product p in Model) {
    <div class="row">
      <div class="col-1">@p.Id</div>
      <div class="col">@p.Name</div>
      <div class="col">@p.Category.Name</div>
      <div class="col text-right">@p.PurchasePrice</div>
      <div class="col text-right">@p.RetailPrice</div>
    </div>
  }
</div>
```

Элемент `button` отправляет HTTP-запрос `POST`, результатом которого будет очистка БД и ее заполнение стандартными категориями и товарами `SportsStore`.

Подготовка базы данных

Чтобы подготовить БД, запустите в папке проекта `SportsStore` команды, приведенные в листинге 9.7. Они добавят новую миграцию, которая отразит изменения в модели данных, после чего удалят и воссоздадут БД.

Листинг 9.7. Команды для подготовки базы данных

```
dotnet ef migrations add Customer
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Seed Data (Начальные данные) и затем на кнопке Production Seed (Заполнить производственными данными). Категории и товары добавятся в БД и отобразятся, как показано на рис. 9.1.

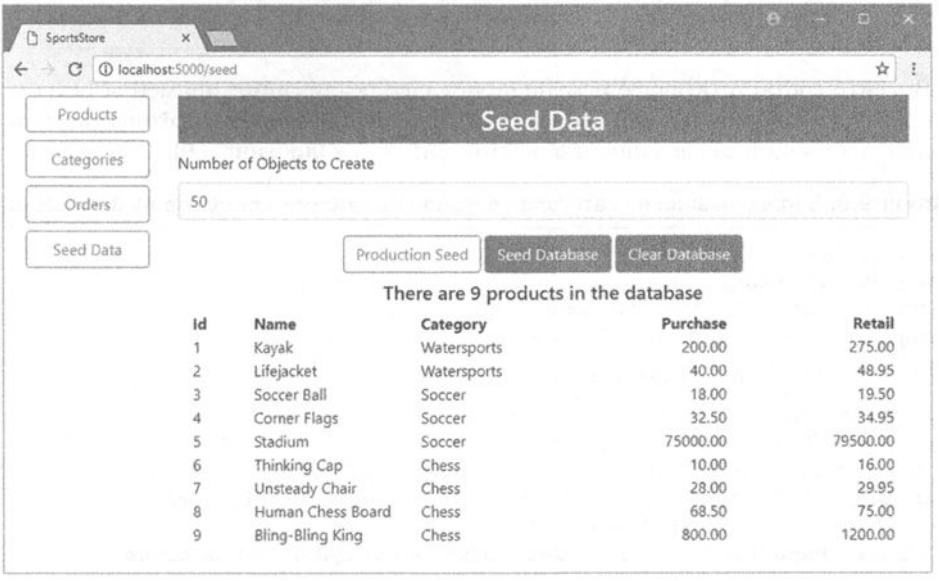

Рис. 9.1. Выполнение примера приложения

Отображение товаров для покупателя

В последующих разделах будет добавлена поддержка для отображения списка товаров пользователю, позволяя ему фильтровать товары по категориям и просматривать товары, доступные для покупки. Поддержка будет строиться на основе средств, созданных в предшествующих главах.

Подготовка модели данных

Чтобы приступить к части приложения, связанной с интерфейсом для покупателей, добавьте возможность запрашивания объектов `Product` по их категории, начав с интерфейса хранилища (листинг 9.8).

Листинг 9.8. Запрашивание по категории в файле `IRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using SportsStore.Models.Pages;

namespace SportsStore.Models {
  public interface IRepository {
    IEnumerable<Product> Products { get; }

    PagedList<Product> GetProducts(QueryOptions options, long category = 0);

    Product GetProduct(long key);

    void AddProduct(Product product);

    void UpdateProduct(Product product);

    void UpdateAll(Product[] products);

    void Delete(Product product);
  }
}
```

Внесите соответствующее изменение в класс реализации, применяя LINQ-метод `Where()` для запрашивания на основе свойства внешнего ключа, которое ассоциирует объект `Product` со связанным объектом `Category` (листинг 9.9).

Листинг 9.9. Запрашивание по категории в файле `DataRepository.cs` из папки `Models`

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models.Pages;

namespace SportsStore.Models {

  public class DataRepository : IRepository {
    private DataContext context;
    public DataRepository(DataContext ctx) => context = ctx;
    public IEnumerable<Product> Products => context.Products
      .Include(p => p.Category).ToArray();
    public PagedList<Product> GetProducts(QueryOptions options,
        long category = 0) {
      IQueryable<Product> query = context.Products.Include(p => p.Category);
      if (category !=- 0) {
        query = query.Where(p => p.CategoryId == category);
      }
      return new PagedList<Product>(query, options);
    }
    // ...для краткости остальные методы не показаны...
  }
}
```

Интерфейс IQueryable<T> позволяет формировать запрос на основе параметров метода, создавая объект, который будет запрашивать БД, только когда он перечисляется. Это преимущество работы с объектами IQueryable<T>, хотя есть и недостаток — легкость, с которой можно непредумышленно инициировать дублированные запросы.

Создание контроллера Store, представлений и компоновки

Чтобы предоставить контроллер, который будет представлять данные покупателю, добавьте в папку Controllers файл по имени StoreController.cs и поместите в него код, приведенный в листинге 9.10.

Листинг 9.10. Содержимое файла StoreController.cs из папки Controllers

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

namespace SportsStore.Controllers {

  public class StoreController : Controller {
    private IRepository productRepository;
    private ICategoryRepository categoryRepository;

    public StoreController(IRepository prepo, ICategoryRepository catRepo) {
      productRepository = prepo;
      categoryRepository = catRepo;
    }

    public IActionResult Index([FromQuery(Name = "options")]
        QueryOptions productOptions,
        QueryOptions catOptions,
        long category) {
      ViewBag.Categories = categoryRepository.GetCategories(catOptions);
      ViewBag.SelectedCategory = category;
      return View(productRepository.GetProducts(productOptions, category));
    }
  }
}
```

Для управления отображением данных Product и Category используются два объекта QueryOptions. Они применяются для получения объекта PagedList<Product>, который передается представлению как его модель, и объекта PagedList<Category>, добавляемого к ViewBag.

Чтобы обеспечить средствам интерфейса для покупателей компоновку, создайте папку Views/Store и поместите в нее файл по имени _Layout.cshtml с содержимым из листинга 9.11.

Листинг 9.11. Содержимое файла _Layout.cshtml из папки Views/Store

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
```

```
  <title>SportsStore</title>
</head>
<body>
  <div class="container-fluid">
    <div class="row bg-dark p-4 text-white">
      <div class="col-auto"><h4>SPORTS STORE</h4></div>
      <div class="col"></div>
      <div class="col-auto text-right">
        (Cart Goes Here)
      </div>
    </div>
  </div>
  @RenderBody()
</body>
</html>
```

Компоновка представляет стандартный заголовок SportsStore, используемый в большинстве моих книг, с заполнителем вместо сводки по корзине для покупок, которая будет добавлена в приложение позже. Для отображения списка товаров добавьте в папку Views/Store файл по имени Index.cshtml с содержимым, показанным в листинге 9.12.

Листинг 9.12. Содержимое файла Index.cshtml из папки Views/Store

```
@model PagedList<Product>

<div class="container-fluid">
  <div class="row no-gutters">
    <div class="col-auto">
     @Html.Partial("Categories", ViewBag.Categories as PagedList<Category>)
    </div>
    <div class="col">
      <div class="container-fluid">
        <div class="row pt-4 pb-1">
          <div class="col text-center">
            @Html.Partial("Pages", Model)
          </div>
        </div>
        <div class="row pt-1 pb-1">
          <div class="col"></div>
          <div class="col-6 text-center form-group">
            <input form="pageform" type="hidden"
                  name="options.searchpropertyname" value="Name" />
            <input form="pageform" name="options.searchterm"
                  placeholder="Seach..." class="form-control" />
          </div>
          <div class="col">
            <button form="pageform" class="btn btn-secondary"
                   type="submit">Search</button>
          </div>
          <div class="col"></div>
        </div>
        @foreach (Product p in Model) {
```

```
        <div class="row">
          <div class="col">
            <div class="card m-1 p-1 bg-light">
              <div class="bg-faded p-1">
                <h4>
                  @p.Name
                  <span class="badge badge-pill badge-primary"
                       style="float:right">
                    <small>$@p.RetailPrice</small>
                  </span>
                </h4>
              </div>
              <form id="@p.Id" asp-action="AddToCart"
                   asp-controller="Cart" method="post">
                <input type="hidden" name="Id" value="@p.Id" />
                <input type="hidden" name="Name"
                       value="@p.Name" />
                <input type="hidden" name="RetailPrice"
                       value="@p.RetailPrice" />
                <input type="hidden" name="returnUrl" value=
                       "@ViewContext.HttpContext.Request.PathAndQuery()" />
                <span class="card-text p-1">
                  @(p.Description
                    ?? "(No Description Available)")
                  <button type="submit"
                          class="btn btn-success btn-sm pull-right"
                          style="float:right">
                    Add To Cart
                  </button>
                </span>
              </form>
            </div>
          </div>
        </div>
      }
    </div>
  </div>
</div>
</div>
```

В представлении `Index` собрано вместе несколько средств для отображения товаров, включая разбиение на страницы и поддержка поиска. Чтобы отобразить пользователю список категорий, добавьте в папку `Views/Store` файл по имени `Categories.cshtml` с содержимым из листинга 9.13.

Листинг 9.13. Содержимое файла `Categories.cshtml` из папки `Views/Store`

```
@model PagedList<Category>

<div class="container-fluid mt-4">
  <div class="row no-gutters">
    <div class="col mt-1">
      <button form="pageform" name="category" value="0" type="submit"
```

```
            class="btn btn-block @(ViewBag.SelectedCategory == 0
              ? "btn-primary" : "btn-outline-primary")">
        All
      </button>
    </div>
  </div>

  <div class="row no-gutters mt-4"></div>

  <div class="row no-gutters">
    <div class="col mt-1">
      <button form="pageform"
            name="catoptions.currentPage" value="@(Model.CurrentPage -1)"
            class="btn btn-block btn-outline-secondary
              @(!Model.HasPreviousPage ? "disabled" : "")"
             type="submit">
        Previous
      </button>
    </div>
  </div>

  @foreach (Category c in Model) {
    <div class="row no-gutters">
      <div class="col mt-1">
        <button form="pageform" name="category" value="@c.Id"
              type="submit"
              class="btn btn-block @(ViewBag.SelectedCategory == c.Id
                ? "btn-primary" : "btn-outline-primary")">
          @c.Name
        </button>
      </div>
    </div>
  }

  <div class="row no-gutters">
    <div class="col mt-1">
      <button form="pageform"
            name="catoptions.currentPage" value="@(Model.CurrentPage +1)"
            class="btn btn-block btn-outline-secondary
              @(!Model.HasNextPage? "disabled" : "")"
             type="submit">
        Next
      </button>
    </div>
  </div>
</div>
```

Представление `Categories` отображает список доступных категорий и предлагает кнопки Previous (Предыдущая) и Next (Следующая) для листания списка. Элементы `button`, выбирающие категории, применяют HTML-форму по имени `pagesform` для нацеливания на контроллер с помощью значения первичного ключа выбранной категории.

Создание URL возврата

Нам необходимо знать, на какой URL переходить после того, как пользователь выбрал товар. Для облегчения процесса создайте папку `Infrastructure` и добавьте в нее файл класса по имени `UrlExtensions.cs` с кодом, приведенным в листинге 9.14.

Листинг 9.14. Содержимое файла `UrlExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

  public static class UrlExtensions {

    public static string PathAndQuery(this HttpRequest request) =>
      request.QueryString.HasValue
        ? $"{request.Path}{request.QueryString}"
        : request.Path.ToString();
  }
}
```

В классе `UrlExtensions` определяется расширяющий метод `PathAndQuery()`, который использовался в элементе `form` в листинге 9.13. Чтобы включить возможность применения этого расширяющего метода в представлении, добавьте в файл импортирования представлений оператор, показанный в листинге 9.15.

Листинг 9.15. Добавление пространства имен в файле `_ViewImports.cshtml` из папки `Views`

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

Тестирование контроллера `Store`

Для просмотра результатов внесенных изменений запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000/store`. Вы увидите список товаров, который можно фильтровать по категориям (рис. 9.2).

Совет. Можете взглянуть, как контроллер `Store` справляется с крупными объемами данных, перейдя по ссылке `http://localhost:5000/seed` и сгенерировав тестовые данные.

Добавление корзины для покупок

Следующий шаг связан с добавлением поддержки для выбора товаров и их сохранения в корзине, которую затем можно применять для оформления заказа. В последующих разделах приложение будет сконфигурировано для сохранения данных сеанса и использования этого в качестве временного хранилища выбранных товаров.

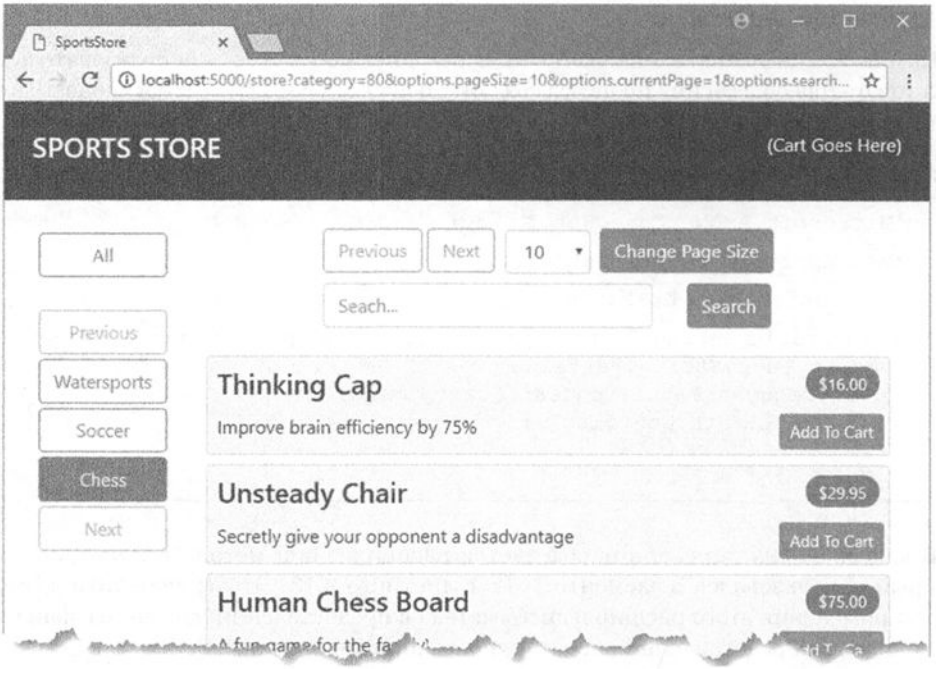

Рис. 9.2. Отображение товаров

Включение постоянства данных сеанса

Некоторые функциональные средства, необходимые для завершения приложения `SportsStore`, нуждаются в сохранении данных между HTTP-запросами, что будет делаться с применением средства данных сеанса ASP.NET Core, сконфигурированного на сохранение своих данных с помощью Entity Framework Core. Чтобы добавить пакет, требующийся для конфигурирования БД сеансов, щелкните правой кнопкой мыши на элементе проекта `SportsStore` в окне Solution Explorer, выберите в контекстном меню пункт Edit SportsStore.csproj (Редактировать `SportsStore.csproj`) и внесите изменения, выделенные в листинге 9.16.

Листинг 9.16. Добавление пакета в файле `SportsStore.csproj` из папки `SportsStore`

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

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

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
    <DotNetCliToolReference
```

```
    Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version="2.0.0" />
  <DotNetCliToolReference
    Include="Microsoft.Extensions.Caching.SqlConfig.Tools"
    Version="2.0.0" />
 </ItemGroup>
</Project>
```

Сохраните изменения, после чего Visual Studio загрузит и установит новый пакет. Откройте новое окно командной строки и выполните в папке проекта SportsStore команду из листинга 9.17 для создания БД сеансов.

Внимание! У вас может возникнуть соблазн создать собственное средство данных сеанса, но не следует недооценивать объем работы, которую придется выполнить, особенно для обеспечения периодической очистки БД от истекших сеансов. Лучше используйте свое время и навыки для создания средств, нужных вашему приложению, и не изобретайте заново то, что уже предлагается компанией Microsoft.

Листинг 9.17. Создание БД сеансов

```
dotnet sql-cache create "Server=(localdb)\MSSQLLocalDB;Database=SportsS
tore" "dbo"
"SessionData"
```

Команда dotnet sql-cache create осуществляет подготовку БД сеансов, но неудобна в работе, поскольку она не читает свою конфигурацию из файла appsettings.json, т.е. аргументы должны набираться внимательно. Первый аргумент — это строка подключения к БД, второй аргумент — имя схемы (dbo по умолчанию) и третий аргумент — имя таблицы, подлежащей добавлению в БД, для которого указано SessionData.

Конфигурирование сеансов в приложении

Чтобы включить сеансы в БД, добавьте в класс Startup операторы, выделенные полужирным в листинге 9.18.

Листинг 9.18. Включение сеансов в файле Startup.cs из папки SportsStore

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

namespace SportsStore {
  public class Startup {
```

```
    public Startup(IConfiguration config) => Configuration = config;
    public IConfiguration Configuration { get; }
    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddTransient<IRepository, DataRepository>();
      services.AddTransient<ICategoryRepository, CategoryRepository>();
      services.AddTransient<IOrdersRepository, OrdersRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
        options.UseSqlServer(conString));
      services.AddDistributedSqlServerCache(options => {
        options.ConnectionString = conString;
        options.SchemaName = "dbo";
        options.TableName = "SessionData";
      });
      services.AddSession(options => {
        options.Cookie.Name = "SportsStore.Session";
        options.IdleTimeout = System.TimeSpan.FromHours(48);
        options.Cookie.HttpOnly = false;
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseSession();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Средство сеансов будет сохранять только значения `string`. Для облегчения работы с этим средством добавьте в папку `Infrastructure` файл класса по имени `SessionExtensions.cs` и поместите в него код из листинга 9.19.

Листинг 9.19. Содержимое файла `SessionExtensions.cs` из папки `Infrastructure`

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace SportsStore.Infrastructure {

  public static class SessionExtensions {

    public static void SetJson(this ISession session, string key, object value) {
      session.SetString(key, JsonConvert.SerializeObject(value));
    }
    public static T GetJson<T>(this ISession session, string key) {
      var sessionData = session.GetString(key);
      return sessionData == null
        ? default(T) : JsonConvert.DeserializeObject<T>(sessionData);
    }
  }
}
```

В классе SessionExtensions определены расширяющие методы, которые сериализируют объекты в формат JSON и восстанавливают их снова, позволяя легко сохранять простые объекты как данные сеанса.

Создание класса модели Cart

Для представления выбранных покупателем товаров добавьте в папку Models файл класса по имени Cart.cs и определите в нем класс, как показано в листинге 9.20.

Листинг 9.20. Содержимое файла Cart.cs из папки Models

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

namespace SportsStore.Models {

  public class Cart {
    private List<OrderLine> selections = new List<OrderLine>();

    public Cart AddItem(Product p, int quantity) {
      OrderLine line = selections
        .Where(l => l.ProductId == p.Id).FirstOrDefault();
      if (line != null) {
        line.Quantity += quantity;
      } else {
        selections.Add(new OrderLine {
          ProductId = p.Id,
          Product = p,
          Quantity = quantity
        });
      }
      return this;
    }

    public Cart RemoveItem(long productId) {
      selections.RemoveAll(l => l.ProductId == productId);
      return this;
    }

    public void Clear() => selections.Clear();

    public IEnumerable<OrderLine> Selections { get => selections; }
  }
}
```

Класс Cart управляет коллекцией объектов OrderLine, которая представляет выбранные товары и может быть легко сохранена в БД при создании заказа.

Создание контроллера и представления

Чтобы предоставить логику, поддерживающую работу с объектами Cart, добавьте в папку Controllers файл по имени CartController.cs и поместите в него код из листинга 9.21.

Листинг 9.21. Содержимое файла CartController.cs из папки Controllers

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

```
using SportsStore.Infrastructure;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Http;
using System.Linq;

namespace SportsStore.Controllers {

  [ViewComponent(Name = "Cart")]
  public class CartController : Controller {
    private IRepository productRepository;
    private IOrdersRepository ordersRepository;

    public CartController(IRepository prepo, IOrdersRepository orepo) {
      productRepository = prepo;
      ordersRepository = orepo;
    }

    public IActionResult Index(string returnUrl) {
      ViewBag.returnUrl = returnUrl;
      return View(GetCart());
    }

    [HttpPost]
    public IActionResult AddToCart(Product product, string returnUrl) {
      SaveCart(GetCart().AddItem(product, 1));
      return RedirectToAction(nameof(Index), new { returnUrl });
    }

    [HttpPost]
    public IActionResult RemoveFromCart(long productId, string returnUrl)
    {
      SaveCart(GetCart().RemoveItem(productId));
      return RedirectToAction(nameof(Index), new { returnUrl });
    }

    public IActionResult CreateOrder() {
      return View();
    }

    [HttpPost]
    public IActionResult CreateOrder(Order order) {
      order.Lines = GetCart().Selections.Select(s => new OrderLine {
        ProductId = s.ProductId,
        Quantity = s.Quantity
      }).ToArray();
      ordersRepository.AddOrder(order);
      SaveCart(new Cart());
      return RedirectToAction(nameof(Completed));
    }

    public IActionResult Completed() => View();

    private Cart GetCart() =>
      HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();

    private void SaveCart(Cart cart) =>
      HttpContext.Session.SetJson("Cart", cart);

    public IViewComponentResult Invoke(ISession session) {
      return new ViewViewComponentResult() {
```

```
            ViewData = new ViewDataDictionary<Cart>(ViewData,
              session.GetJson<Cart>("Cart"))
          };
        }
      }
    }
```

Контроллер определяет действия, которые добавляют и удаляют элементы из корзины, отображают содержимое корзины и предоставляют покупателю возможность оформить заказ. Ряд методов принимает параметр `returnUrl`, позволяющий пользователю возвратиться к списку товаров, не утрачивая параметры строки запроса, которые конфигурируют разбиение на страницы и фильтрацию по категории. Класс `CartController` также является компонентом представления, который будет применяться для отображения сводки по корзине в компоновке интерфейса покупателей.

Создание представлений

Чтобы снабдить новый контроллер представлением для управления корзиной, создайте папку `Views/Cart` и добавьте в нее файл по имени `Index.cshtml` с содержимым, приведенным в листинге 9.22.

Листинг 9.22. Содержимое файла `Index.cshtml` из папки `Views/Cart`

```
@model Cart
@{
  Layout = "~/Views/Store/_Layout.cshtml";
}

<h2 class="m-3">Your Cart</h2>

<div class="container-fluid">
  <div class="row">
    <div class="col font-weight-bold">Quantity</div>
    <div class="col font-weight-bold">Product</div>
    <div class="col font-weight-bold text-right">Price</div>
    <div class="col font-weight-bold text-right">Subtotal</div>
    <div class="col"></div>
  </div>
  @if (Model.Selections.Count() == 0) {
    <div class="row mt-2"><div class="col-12"><h4>Cart is Empty</h4></div>
    </div>
  } else {
    @foreach (OrderLine line in Model.Selections) {
      <div class="row mt-1">
        <div class="col">@line.Quantity</div>
        <div class="col">@line.Product.Name</div>
        <div class="col text-right">
          $@line.Product.RetailPrice.ToString("f2")
        </div>
        <div class="col text-right">
          $@((line.Product.RetailPrice
            * line.Quantity).ToString("f2"))
        </div>
```

```
        <div class="col">
          <form asp-action="RemoveFromCart">
            <button type="submit" name="productId"
                    value="@line.ProductId"
                    class="btn btn-sm btn-outline-danger">Remove</button>
          </form>
        </div>
      </div>
    }
  }
  <div class="row mt-2">
    <div class="col"></div>
    <div class="col"></div>
    <div class="col text-right font-weight-bold">Total:</div>
    <div class="col text-right font-weight-bold">
      $@(Model.Selections.Sum(l => l.Product.RetailPrice
        * l.Quantity).ToString("f2"))
    </div>
    <div class="col"></div>
  </div>
</div>
<div class="text-center m-2">
  @if (ViewBag.returnUrl != null) {
    <a href="@ViewBag.returnUrl" class="btn btn-outline-primary">
      Continue Shopping
    </a>
  }
  <a asp-action="CreateOrder" class="btn btn-primary">
    Place Order
  </a>
</div>
```

Представление отображает сводку по товарам, выбранным покупателем, и предлагает кнопки, которые позволяют возвратиться к списку товаров или продолжить оформление заказа. Чтобы собрать информацию для создания заказа, добавьте в папку `Views/Cart` файл по имени `CreateOrder.cshtml` с содержимым из листинга 9.23.

Листинг 9.23. Содержимое файла `CreateOrder.cshtml` из папки `Views/Cart`

```
@model Order
@{
  Layout = "~/Views/Store/_Layout.cshtml";
}

<h2 class="m-3">Your Details</h2>

<form asp-action="CreateOrder" method="post" class="m-4">
  <div class="form-group">
    <label>Your Name:</label>
    <input asp-for="CustomerName" class="form-control" />
  </div>
  <div class="form-group">
    <label> Your Address</label>
    <input asp-for="Address" class="form-control" />
  </div>
```

```
  <div class="form-group">
    <label>Your State:</label>
    <input asp-for="State" class="form-control" />
  </div>
  <div class="form-group">
    <label>Your Zip Code:</label>
    <input asp-for="ZipCode" class="form-control" />
  </div>
  <div class="text-center m-2">
    <button type="submit" class="btn btn-primary">Place Order</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Для отображения пользователю сообщения о том, что заказ создан, добавьте в папку `Views/Cart` файл по имени `Completed.cshtml`, содержимое которого показано в листинге 9.24.

Листинг 9.24. Содержимое файла `Completed.cshtml` из папки `Views/Cart`

```
@{
  Layout = "~/Views/Store/_Layout.cshtml";
}
<div class="text-center m-4">
  <h2>Thanks!</h2>
  <p>Thanks for placing your order.</p>
  <p>We'll ship your goods as soon as possible.</p>
  <a asp-action="Index" asp-controller="Store"
     class="btn btn-primary">
    OK
  </a>
</div>
```

Чтобы создать представление для виджета (графического элемента) со сводкой по корзине, создайте папку `Views/Shared/Components/Cart` и добавьте в нее файл по имени `Cart.cshtml` с содержимым, приведенным в листинге 9.25.

Листинг 9.25. Содержимое файла `Default.cshtml` из папки `Views/Shared/Components/Cart`

```
@model Cart

@if (Model?.Selections?.Count() > 0) {
  <div>@Model.Selections.Count() items,
    $@(Model.Selections.Sum(l => l.Quantity
      * l.Product.RetailPrice).ToString("f2"))
  </div>
  if (ViewContext.RouteData.Values["controller"] as string != "Cart") {
    <a asp-action="Index" asp-controller="Cart"
       class="btn btn-sm btn-light">
      Checkout
    </a>
  }
}
```

Представление Default отображает количество элементов в корзине и их общую стоимость. Также имеется кнопка, которая переместит на контроллер Cart, если представление было визуализировано другим контроллером. Чтобы отобразить виджет корзины, с помощью метода Component.InvokeAsync() добавьте компонент представления в компоновку, используемую для функциональных средств магазина (листинг 9.26).

Листинг 9.26. Добавление элемента в файле _Layout.cshtml из папки Views/Store

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
  <title>SportsStore</title>
</head>
<body>
  <div class="container-fluid">
    <div class="row bg-dark p-4 text-white">
      <div class="col-auto"><h4>SPORTS STORE</h4></div>
      <div class="col"></div>
      <div class="col-auto text-right">
        @await Component.InvokeAsync("Cart", Context.Session)
      </div>
    </div>
  </div>
  @RenderBody()
</body>
</html>
```

Тестирование процесса оформления заказа

Для тестирования процесса оформления заказа запустите приложение с применением команды dotnet run и перейдите по ссылке http://localhost:5000/store. Щелкните на кнопке Add to Cart (Добавить в корзину) для одного или большего числа товаров и затем на кнопке Place Order (Разместить заказ). Введите детальные сведения о покупателе и щелкните на кнопке Place Order; вы увидите сообщение о том, что заказ размещен. Вся последовательность проиллюстрирована на рис. 9.3.

Распространенные проблемы и их решения

Материал главы был сосредоточен в основном на построении функциональных средств приложения с использованием MVC и нескольких добавлений, которые вполне вероятно могут вызвать проблемы с Entity Framework Core.

Щелчок на кнопке страницы управляет ошибочным типом данных

Если вы щелкаете на кнопке для изменения страницы категорий, например, но изменяется страница товаров, то вероятная причина заключается в том, что HTML-формы в представлениях обновляют ошибочный объект PageOptions.

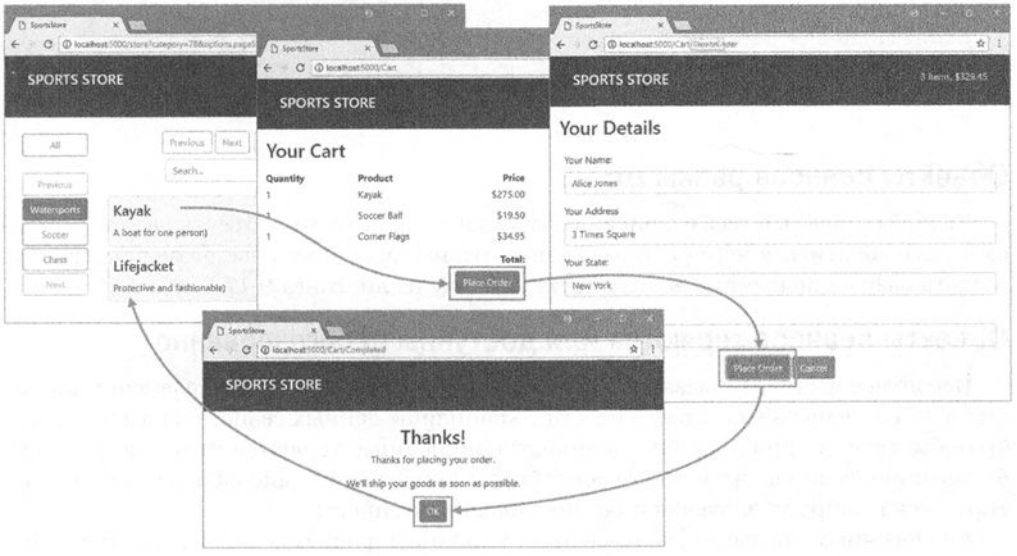

Рис. 9.3. Размещение заказа

Обратите пристальное внимание на имена параметров в методах действий и удостоверьтесь в том, что применяете эти имена в качестве префиксов в атрибутах `name` элементов HTML. В случае сомнений воспользуйтесь инструментами разработчика в браузере, доступными по нажатию клавиши <F12>, чтобы посмотреть, какие значения форм отправляются приложению.

Щелчок на кнопке страницы не имеет никакого эффекта

Самая распространенная причина отсутствия эффекта у кнопок — пропуск атрибута `form`, который применяется для ассоциирования HTML-элемента с формой. Эта особенность HTML 5 зачастую вызывает путаницу, поскольку разработчики еще не привыкли расширять форму за пределами элемента `form`.

Исключение "Cannot Insert Explicit Value for Identity Column" ("Не удается вставить явное значение для идентичности")

Хотя это исключение было описано ранее, стоит упомянуть о нем еще раз, потому что оно возникает часто. В контексте настоящей главы вероятной причиной является запрашивание у БД объектов и последующее их сохранение в виде данных сеанса, как ранее делалось с объектами `Product`, ассоциированными с `OrderLine`. Объекты, к которым вы обратились, уже имеют первичные ключи, а исключение генерируется из-за того, что Entity Framework Core пытается добавить их в БД как новые объекты.

Когда вы позже получаете объект из данных сеанса и сохраняете его в БД, то должны убедиться, что удалили ссылки на связанные данные, которые запрашивали ранее. В случае объектов `OrderLine`, сохраняемых внутри `Cart` в виде данных сеанса и затем сохраняемых в БД с объектом `Order`, вот как удалялись ссылки на объекты `Product`, чтобы получить в итоге чистый набор объектов `OrderLine`:

```
...
order.Lines = GetCart().Selections.Select(s => new OrderLine {
  ProductId = s.ProductId, Quantity = s.Quantity
}).ToArray();
...
```

Объекты сеансов равны null

Если вы сталкиваетесь с ошибкой, указывающей на то, что объекты, к которым вы ожидаете обратиться через данные сеанса, равны `null`, тогда вы, возможно, забыли создать базу данных сеансов, используя команду из листинга 9.17.

Объекты сеансов теряются или доступны несогласованно

Вероятнее всего, это вызвано конфигурированием средства сеансов для сохранения данных сеансов в памяти, а не в БД. Хранилище данных сеансов в памяти может быть быстрее, но при перезапуске приложения данные теряются, что особенно проблематично, если вы применяете контейнеры приложений наподобие Docker и адаптируетесь к запросу, запуская и останавливая контейнеры.

Аналогичным образом использование хранилища данных сеансов в памяти может привести к получению несогласованных данных сеансов, когда функционирует множество экземпляров приложения MVC, но вы не сконфигурировали сеть так, чтобы HTTP-запросы от конкретного клиента обрабатывались всегда тем же самым экземпляром приложения MVC. Столкнувшись с любой из указанных проблем, подумайте о хранении данных сеансов в БД, как демонстрировалось в этой главе.

Резюме

В главе работа над приложением `SportsStore` продолжилась добавлением средств интерфейса для покупателей. Был создан список товаров, который пользователь может пролистывать, искать в нем товары или фильтровать их по категориям. Выбранные товары добавляются в корзину, которая затем может применяться для оформления заказа, сохраняемого в БД. Большинство функциональных средств, добавленных в главе, используют инфраструктуру MVC для создания надстройки поверх фундамента Entity Framework Core, образованного в предшествующих главах. Такой шаблон вы будете видеть и в собственных проектах — большой объем первоначальной конфигурации модели данных и кода, а затем набор средств пользовательского интерфейса, которые быстро встают на свои места. В следующей главе проект приложения `SportsStore` завершается созданием веб-службы REST.

ГЛАВА 10

SportsStore: создание веб-службы REST

Веб-службы удобны при снабжении данными клиентских приложений, которые пишутся с использованием таких инфраструктур, как Angular или React. Клиентские приложения обычно запускаются в браузерах и не требуют HTML-содержимого, которое предоставляет остаток приложения SportsStore. Взамен они взаимодействуют с приложением ASP.NET Core MVC, применяя HTTP-запросы, и получают данные, сформатированные по стандарту JSON (JavaScript Object Notation — система обозначений для объектов JavaScript).

В этой главе приложение SportsStore будет завершено путем добавления веб-службы REST, которая способна предоставить веб-клиентам доступ к данным приложения. Инфраструктура ASP.NET Core MVC располагает великолепной поддержкой для создания веб-служб REST, но когда используется Entity Framework Core, необходимо проявить особое внимание, чтобы добиться правильных результатов.

Не существует каких-то жестких правил относительно того, как должны работать веб-службы, но самый распространенный подход предусматривает принятие патерна REST (REpresentational State Transfer — передача состояния представления). Официальная спецификация REST отсутствует, и нет единодушия в плане того, из чего состоит веб-служба REST, но есть ряд основных мыслей, широко применяемых для веб-служб.

Главная предпосылка веб-службы REST — и единственный аспект, получивший полное согласие, — заключается в том, что веб-служба определяет API-интерфейс посредством сочетания адресов URL и методов HTTP, таких как GET и POST. Метод HTTP устанавливает тип операции, а URL указывает объект или объекты данных, к которым эта операция применяется.

Подготовительные шаги

Мы продолжим использовать проект SportsStore, который был создан в главе 4 и наращивался в последующих главах. Находясь в папке проекта SportsStore, выполните команды из листинга 10.1, чтобы сбросить БД.

Листинг 10.1. Сброс БД примера приложения

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Seed Data (Начальные данные) и затем на кнопке Production Seed (Заполнить производственными данными). БД заполнится небольшим количеством товаров (рис. 10.1).

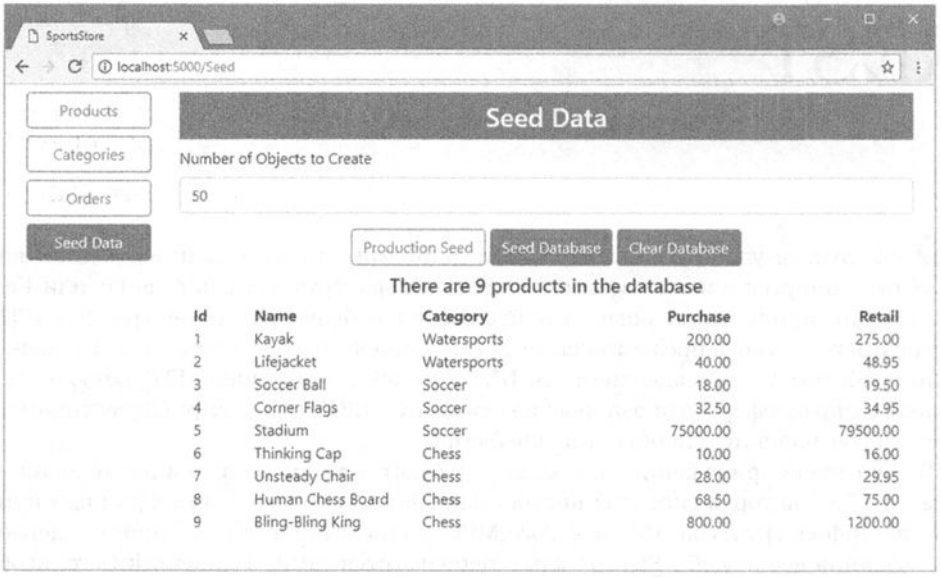

Рис. 10.1. Выполнение примера приложения

Совет. Проект `SportsStore` и проекты для остальных глав книги доступны для загрузки в хранилище GitHub по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Создание веб-службы

В последующих разделах будет построена простая веб-служба, которая обеспечит доступ к данным `Product`, сохраненным приложением `SportsStore`.

Создание хранилища

При добавлении веб-службы в приложение разумно создать отдельное хранилище, поскольку запросы, выполняемые клиентскими приложениями, могут отличаться от запросов обычного приложения ASP.NET Core MVC. Для веб-службы `SportsStore` добавьте в папку `Models` файл по имени `IWebServiceRepository.cs` и определите в нем интерфейс, как показано в листинге 10.2.

Листинг 10.2. Содержимое файла IWebServiceRepository.cs из папки Models

```
namespace SportsStore.Models {
  public interface IWebServiceRepository {
    object GetProduct(long id);
  }
}
```

Мы начинаем с единственного метода GetProduct(), который будет принимать значение первичного ключа и возвращать из БД соответствующий объект Product. Метод GetProduct() возвращает результат типа object, а не Product, так что можно продемонстрировать, каким образом представлять инфраструктуре Entity Framework Core данные из веб-службы.

Для класса реализации хранилища добавьте в папку Models файл по имени WebServiceRepository.cs и определите в нем класс, приведенный в листинге 10.3.

Листинг 10.3. Содержимое файла WebServiceRepository.cs из папки Models

```
using System.Linq;

namespace SportsStore.Models {
  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;

    public WebServiceRepository(DataContext ctx) => context = ctx;

    public object GetProduct(long id) {
      return context.Products.FirstOrDefault(p => p.Id == id);
    }
  }
}
```

Класс WebServiceRepository реализует метод GetProduct() за счет использования LINQ-метода FirstOrDefault() для нахождения объекта, хранящегося в БД, который имеет указанное значение Id. При создании веб-службы важно позаботиться о запросах в отношении несуществующих данных, из-за чего и применялся метод FirstOrDefault().

Чтобы зарегистрировать хранилище и класс его реализации, добавьте в класс Startup оператор, выделенный в листинге 10.4.

Листинг 10.4. Конфигурирование хранилища в файле Startup.cs из папки SportsStore

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

```
using Microsoft.Extensions.Configuration;

namespace SportsStore {
  public class Startup {

    public Startup(IConfiguration config) => Configuration = config;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc();
      services.AddTransient<IRepository, DataRepository>();
      services.AddTransient<ICategoryRepository, CategoryRepository>();
      services.AddTransient<IOrdersRepository, OrdersRepository>();
      services.AddTransient<IWebServiceRepository, WebServiceRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
        options.UseSqlServer(conString));

      services.AddDistributedSqlServerCache(options => {
        options.ConnectionString = conString;
        options.SchemaName = "dbo";
        options.TableName = "SessionData";
      });
      services.AddSession(options => {
        options.Cookie.Name = "SportsStore.Session";
        options.IdleTimeout = System.TimeSpan.FromHours(48);
        options.Cookie.HttpOnly = false;
      });
    }

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

Создание контроллера API

Инфраструктура ASP.NET Core MVC облегчает добавление веб-служб в приложение, используя стандартные средства контроллеров. Добавьте в папку `Controllers` файл класса по имени `ProductValuesController.cs` и поместите в него код из листинга 10.5.

Листинг 10.5. Содержимое файла `ProductValuesController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {
```

```
  [Route("api/products")]
  public class ProductValuesController : Controller {
    private IWebServiceRepository repository;

    public ProductValuesController(IWebServiceRepository repo)
      => repository = repo;

    [HttpGet("{id}")]
    public object GetProduct(long id) {
      return repository.GetProduct(id) ?? NotFound();
    }
  }
}
```

Новый класс контроллера имеет имя `ProductValuesController`, которое следует соглашению о включении слова `Values` в имя, чтобы указать, что контроллер будет возвращать своим клиентам данные, а не HTML-разметку. Еще одно соглашение в отношении контроллеров веб-служб связано с созданием отдельной части схемы маршрутизации, специально предназначенной для обработки запросов данных. Наиболее распространенный способ предполагает создание URL веб-службы, начинающихся со строки `/api`, за которой следует форма множественного числа для имени типа данных, обрабатываемого веб-службой. В случае веб-службы, обрабатывающей объекты `Product`, это означает, что HTTP-запросы должны отправляться на URL вида `/api/products`, который был сконфигурирован с применением атрибута `Route`:

```
...
[Route("api/products")]
...
```

Единственным действием, определенным в контроллере в текущий момент, является метод `GetProduct()`, который возвращает одиночный объект `Product` на основе первичного ключа — значения, присвоенного его свойству `Id`. Метод действия декорирован атрибутом `HttpGet`, который позволит инфраструктуре ASP.NET Core MVC использовать это действие для обработки HTTP-запросов `GET`:

```
...
[HttpGet("{id}")]
public Product GetProduct(long id) {
...
```

Аргумент атрибута расширяет схему URL, определенную атрибутом `Route`, так чтобы метод `GetProduct()` мог быть достигнут по URL в форме `/api/products/{id}`. Методы действий для веб-служб возвращают объекты .NET, которые автоматически сериализируются и отправляются клиенту.

Чтобы воспрепятствовать сериализации веб-службой ответа `null`, когда запрошен несуществующий объект, метод действия применяет операцию объединения с `null` для вызова метода `NotFound()`:

```
...
return repository.GetProduct(id) ?? NotFound();
...
```

В результате возвратится код состояния `404 - Not Found` (404 — не найдено), который сигнализирует клиента о том, что запрос не мог быть удовлетворен.

Тестирование веб-службы

Для тестирования новой веб-службы запустите приложение, используя `dotnet run`. Откройте новое окно PowerShell и выполните команду из листинга 10.6, чтобы отправить HTTP-запрос `GET` контроллеру API.

На заметку! Повсюду в главе для эмуляции запросов от клиентского приложения применяется PowerShell-команда `Invoke-RestMethod`.

Листинг 10.6. Тестирование примера веб-службы

```
Invoke-RestMethod http://localhost:5000/api/products/1 -Method GET |
ConvertTo-Json
```

Запрос HTTP отправляется методу `GetProduct()` контроллера `ProductValues`, который использует метод `Find()` для извлечения объекта `Product` из БД. Объект `Product` сериализируется в формат JSON и возвращается браузеру, который отобразит полученные данные:

```
{
  "id":1,
  "name":"Kayak",
  "description":"A boat for one person",
  "purchasePrice":200.00,
  "retailPrice":275.00,
  "categoryId":1,
  "category":null
}
```

Обратите внимание, что навигационное свойство `category` установлено в `null`, потому что у инфраструктуры Entity Framework Core не запрашивалась загрузка связанных данных для объекта `Product`.

Проецирование результата для исключения навигационных свойств, равных `null`

Отправка клиентскому приложению данных, которые содержат свойства `null`, может привести к путанице, т.к. неясно, отсутствуют связанные данные или они просто не были включены в ответ. О модели данных вы уже знаете достаточно много, чтобы понять, что значение `categoryId` указывает на наличие связанных данных, но ожидать от клиентского приложения реализации такого различения проблематично, особенно если оно разрабатывается другой командой программистов. Если вы не хотите включать свойства связанных данных, то можете избежать путаницы за счет применения LINQ для проецирования результата, который исключает свойства внешнего ключа и навигационные свойства, как показано в листинге 10.7.

Листинг 10.7. Исключение свойств в файле `WebServiceRepository.cs` из папки `Models`

```
using System.Linq;

namespace SportsStore.Models {
```

```
  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;
    public WebServiceRepository(DataContext ctx) => context = ctx;

    public object GetProduct(long id) {
      return context.Products
        .Select(p => new { Id = p.Id, Name = p.Name,
          Description = p.Description, PurchasePrice = p.PurchasePrice,
          RetailPrice = p.RetailPrice})
        .FirstOrDefault(p => p.Id == id);
    }
  }
}
```

Метод `Select()` из LINQ используется для выбора свойств, подлежащих включению в результат, а метод `FirstOrDefault()` — для выбора объекта с указанным значением первичного ключа. Перезапустите приложение с применением `dotnet run` и в отдельном окне PowerShell выполните команду, представленную в листинге 10.8.

Листинг 10.8. Запрашивание объекта `Product`

```
Invoke-RestMethod http://localhost:5000/api/products/1
      -Method GET | ConvertTo-Json
```

Результатом запроса являются следующие данные JSON, из которых исключены свойства связанных данных:

```
{
  "id":1,
  "name":"Kayak",
  "description":"A boat for one person",
  "purchasePrice":200.00,
  "retailPrice":275.00
}
```

Просмотрите журнальные сообщения, сгенерированные приложением, и вы увидите, что инфраструктура Entity Framework Core запросила из БД только указанные свойства:

```
...
SELECT TOP(1) [p].[Id], [p].[Name], [p].[Description], [p].[PurchasePrice],
  [p].[RetailPrice]
FROM [Products] AS [p]
WHERE [p].[Id] = @__id_0
...
```

Включение связанных данных в ответ веб-службы

Если вы хотите включить связанные данные в ответ веб-службы, тогда потребуется проявить осторожность. Для демонстрации проблемы измените запрос, использующий хранилище, чтобы в нем применялся метод `Include()` с целью выбора объекта `Category`, с которым ассоциирован объект `Product` (листинг 10.9).

Листинг 10.9. Включение связанных данных в файле `WebServiceRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;

    public WebServiceRepository(DataContext ctx) => context = ctx;

    public object GetProduct(long id) {
      return context.Products.Include(p => p.Category)
        .FirstOrDefault(p => p.Id == id);
    }
  }
}
```

Чтобы увидеть, какую проблему скрывает этот код, запустите приложение, используя `dotnet run`, и в отдельном окне PowerShell выполните команду из листинга 10.10.

Листинг 10.10. Запрашивание связанных данных

```
Invoke-RestMethod http://localhost:5000/api/products/1
      -Method GET | ConvertTo-Json
```

Вместо данных JSON отобразится сообщение об ошибке:

```
Invoke-RestMethod : Unable to read data from the transport connection:
The connection was closed.
Invoke-RestMethod : Не удается прочитать данные из транспортного
подключения: Подключение было закрыто.
```

При работе с сериализацией инфраструктура ASP.NET Core MVC применяет пакет под названием Json.NET и для выявления причины ошибки потребуется внести изменение в конфигурацию (листинг 10.11).

Листинг 10.11. Изменение конфигурации сериализатора 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.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Newtonsoft.Json;

namespace SportsStore {
  public class Startup {
```

```
    public Startup(IConfiguration config) => Configuration = config;

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services) {
      services.AddMvc().AddJsonOptions(opts =>
        opts.SerializerSettings.ReferenceLoopHandling
          = ReferenceLoopHandling.Serialize);
      services.AddTransient<IRepository, DataRepository>();
      services.AddTransient<ICategoryRepository, CategoryRepository>();
      services.AddTransient<IOrdersRepository, OrdersRepository>();
      services.AddTransient<IWebServiceRepository, WebServiceRepository>();
      string conString = Configuration["ConnectionStrings:DefaultConnection"];
      services.AddDbContext<DataContext>(options =>
        options.UseSqlServer(conString));
      services.AddDistributedSqlServerCache(options => {
        options.ConnectionString = conString;
        options.SchemaName = "dbo";
        options.TableName = "SessionData";
      });
      services.AddSession(options => {
        options.Cookie.Name = "SportsStore.Session";
        options.IdleTimeout = System.TimeSpan.FromHours(48);
        options.Cookie.HttpOnly = false;
      });
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseSession();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Перезапустите приложение, используя `dotnet run`. Вместо применения PowerShell откройте окно браузера и посетите URL вида `http://localhost:5000/api/products/1`. Когда браузер запросит данные, вы увидите, что приложение сообщит о следующей ошибке и завершится:

```
...
Process is terminating due to StackOverflowException
Процесс завершается из-за исключения StackOverflowException
...
```

Отображаемое браузером содержимое показывает, что произошло. Хотя запрошенный URL нацелен на одиночный объект `Product` и связанный с ним объект `Category`, прежде чем потерпеть отказ, приложение отправило много данных:

```
...
{"id":1,"name":"Kayak","description":"A boat for one person",
  "purchasePrice":200.00,"retailPrice":275.00,"categoryId":1,
  "category": {"id":1,"name":"Watersports",
    "description":"Make a splash",
```

```
"products":[{"id":1,"name":"Kayak",
  "description":"A boat for one person",
  "purchasePrice":200.00,"retailPrice":275.00,
  "categoryId":1,
  "category":{"id":1,"name":"Watersports",
    "description":"Make a splash",
    "products":[{"id":1,"name":"Kayak","description":"A boat for one
...
```

Образовался бесконечный цикл, где навигационное свойство `Category` объекта `Product` указывает на связанный объект `Category`, чье навигационное свойство `Products` включает объект `Product` и т.д. Он обусловлен средством инфраструктуры Entity Framework Core, называемым *исправлением*, которое использует объекты, полученные из БД, в качестве значений для навигационных свойств. В главе 14 процесс исправления рассматривается подробно и объясняется, когда он может быть полезен, но для веб-служб REST такое средство вызывает проблемы, потому что сериализатор JSON продолжает следовать по навигационным свойствам до тех пор, пока в приложении не возникнет переполнение стека. Изменение конфигурации в листинге 10.11 сообщает сериализатору JSON о необходимости продолжать следовать по ссылкам, даже если объект уже сериализирован. Стандартное поведение предусматривает генерацию исключения, когда обнаруживается цикл, что и стало причиной возникновения ошибки при выполнении команды из листинга 10.10.

Избегание циклических ссылок в связанных данных

Отключить средство исправления невозможно, поэтому наилучшим решением, позволяющим избежать бесконечных циклов в связанных данных, будет установка навигационного свойства в `null`, прежде чем оно достигнет сериализатора JSON. Тем самым создается динамический тип, и значения `null` в результат не помещаются — именно такой подход принят в листинге 10.12.

Листинг 10.12. Избегание бесконечного цикла в файле `WebServiceRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;

    public WebServiceRepository(DataContext ctx) => context = ctx;

    public object GetProduct(long id) {
      return context.Products.Include(p => p.Category)
        .Select(p => new {
          Id = p.Id, Name = p.Name, PurchasePrice = p.PurchasePrice,
          Description = p.Description, RetailPrice = p.RetailPrice,
          CategoryId = p.CategoryId,
          Category = new {
            Id = p.Category.Id,
            Name = p.Category.Name,
            Description = p.Category.Description
          }
        })
```

```
            .FirstOrDefault(p => p.Id == id);
        }
    }
}
```

Запустите приложение с применением `dotnet run` и выполните в отдельном окне PowerShell команду из листинга 10.13.

Листинг 10.13. Запрашивание связанных данных

```
Invoke-RestMethod http://localhost:5000/api/products/1
      -Method GET | ConvertTo-Json
```

Команда производит следующий результат JSON, который включает все связанные данные, но не содержит циклическую ссылку:

```
...
{
  "id": 1,
  "name": "Kayak",
  "purchasePrice": 200.00,
  "description": "A boat for one person",
  "retailPrice": 275.00,
  "categoryId": 1,
  "category":
  {
    "id": 1,
    "name": "Watersports",
    "description": "Make a splash"
  }
}
...
```

Запрашивание множества объектов

При построении запросов для множества объектов важно ограничивать объем данных, отправляемых клиентскому приложению. Если вы просто возвратите все объекты из БД, то увеличите затраты на запуск своего приложения и даже можете перегрузить данными клиентское приложение. Добавьте в интерфейс хранилища веб-службы метод, который позволит клиенту указывать начальный индекс для результатов и количество требующихся объектов (листинг 10.14).

Листинг 10.14. Добавление метода в файле `IWebServiceRepository.cs` из папки `Models`

```
namespace SportsStore.Models {
  public interface IWebServiceRepository {
    object GetProduct(long id);
    object GetProducts(int skip, int take);
  }
}
```

Добавьте этот метод в класс реализации и воспользуйтесь приемом из предыдущего раздела, чтобы избежать циклических ссылок в связанных данных (листинг 10.15).

Листинг 10.15. Добавление метода в файле `WebServiceRepository.cs` из папки `Models`

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

namespace SportsStore.Models {

  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;

    public WebServiceRepository(DataContext ctx) => context = ctx;

    public object GetProduct(long id) {
      return context.Products.Include(p => p.Category)
        .Select(p => new {
          Id = p.Id, Name = p.Name, PurchasePrice = p.PurchasePrice,
          Description = p.Description, RetailPrice = p.RetailPrice,
          CategoryId = p.CategoryId,
          Category = new {
            Id = p.Category.Id,
            Name = p.Category.Name,
            Description = p.Category.Description
          }
        })
        .FirstOrDefault(p => p.Id == id);
    }

    public object GetProducts(int skip, int take) {
      return context.Products.Include(p => p.Category)
        .OrderBy(p => p.Id)
        .Skip(skip)
        .Take(take)
        .Select(p => new {
          Id = p.Id, Name = p.Name, PurchasePrice = p.PurchasePrice,
          Description = p.Description, RetailPrice = p.RetailPrice,
          CategoryId = p.CategoryId,
          Category = new {
            Id = p.Category.Id,
            Name = p.Category.Name,
            Description = p.Category.Description
          }
        });
    }
  }
}
```

Добавьте к контроллеру веб-службы действие, которое даст клиентам возможность запрашивать множество объектов (листинг 10.16).

Листинг 10.16. Добавление действия в файле `ProductValuesController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  [Route("api/products")]
  public class ProductValuesController : Controller {
    private IWebServiceRepository repository;

    public ProductValuesController(IWebServiceRepository repo)
      => repository = repo;

    [HttpGet("{id}")]
    public object GetProduct(long id) {
      return repository.GetProduct(id) ?? NotFound();
    }

    [HttpGet]
    public object Products(int skip, int take) {
      return repository.GetProducts(skip, take);
    }
  }
}
```

Чтобы протестировать новый метод действия, запустите приложение с применением `dotnet run` и выполните в отдельном окне PowerShell команду, показанную в листинге 10.17.

Листинг 10.17. Запрашивание множества объектов

```
Invoke-RestMethod http://localhost:5000/api/products?skip=2"&"take=2
      -Method GET | ConvertTo-Json
```

Запрос HTTP указывает веб-службе о том, что нужно пропустить первые два объекта и затем возвратить следующие два объекта, что производит такие результаты:

```
...
{
  "value": [
  {
    "id": 3,
    "name": "Soccer Ball",
    "purchasePrice": 18.00,
    "description": "FIFA-approved size and weight",
    "retailPrice": 19.50,
    "categoryId": 2,
    "category": {
      "id": 2,
      "name": "Soccer",
      "description": "The World\u0027s Favorite Game"
    }
  },
```

```
    {
      "id": 4,
      "name": "Corner Flags",
      "purchasePrice": 32.50,
      "description": "Give your playing field a professional touch",
      "retailPrice": 34.95,
      "categoryId": 2,
      "category": {
        "id": 2,
        "name": "Soccer",
        "description": "The World\u0027s Favorite Game"
      }
    }],
    "Count": 2
  }
  ...
```

Завершение веб-службы

Сложность веб-службы, использующей данные Entity Framework Core, кроется в сериализации ответов, как было описано в предыдущем разделе. Остальные стандартные операции следуют тому же самому шаблону, который вы видели в предшествующих главах. Добавьте к интерфейсу хранилища методы, которые позволят сохранять, обновлять и удалять объекты (листинг 10.18).

Листинг 10.18. Добавление методов в файле `IWebServiceRepository.cs` из папки `Models`

```
namespace SportsStore.Models {
  public interface IWebServiceRepository {
    object GetProduct(long id);
    object GetProducts(int skip, int take);
    long StoreProduct(Product product);
    void UpdateProduct(Product product);
    void DeleteProduct(long id);
  }
}
```

Добавьте новые методы в класс реализации хранилища (листинг 10.19).

Листинг 10.19. Добавление методов в файле `WebServiceRepository.cs` из папки `Models`

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

namespace SportsStore.Models {
  public class WebServiceRepository : IWebServiceRepository {
    private DataContext context;
    public WebServiceRepository(DataContext ctx) => context = ctx;
```

```
    public object GetProduct(long id) {
      return context.Products.Include(p => p.Category)
        .Select(p => new {
          Id = p.Id, Name = p.Name, PurchasePrice = p.PurchasePrice,
          Description = p.Description, RetailPrice = p.RetailPrice,
          CategoryId = p.CategoryId,
          Category = new {
            Id = p.Category.Id,
            Name = p.Category.Name,
            Description = p.Category.Description
          }
        })
        .FirstOrDefault(p => p.Id == id);
    }
    public object GetProducts(int skip, int take) {
      return context.Products.Include(p => p.Category)
        .OrderBy(p => p.Id)
        .Skip(skip)
        .Take(take)
        .Select(p => new {
          Id = p.Id, Name = p.Name, PurchasePrice = p.PurchasePrice,
          Description = p.Description, RetailPrice = p.RetailPrice,
          CategoryId = p.CategoryId,
          Category = new {
            Id = p.Category.Id,
            Name = p.Category.Name,
            Description = p.Category.Description
          }
        });
    }
    public long StoreProduct(Product product) {
      context.Products.Add(product);
      context.SaveChanges();
      return product.Id;
    }
    public void UpdateProduct(Product product) {
      context.Products.Update(product);
      context.SaveChanges();
    }
    public void DeleteProduct(long id) {
      context.Products.Remove(new Product { Id = id });
      context.SaveChanges();
    }
  }
}
```

Обратите внимание, что методы `StoreProduct()` возвращают значение первичного ключа, назначенное объекту `Product` сервером баз данных. Клиентские приложения часто поддерживают собственные модели данных, и важно гарантировать, что они располагают всей информацией, которая требуется им для выполнения последующих операций без необходимости в выдаче дополнительных запросов.

Модификация контроллера

Модифицируйте контроллер веб-службы, добавив действия, которые соответствуют новым методам хранилища, как показано в листинге 10.20.

Листинг 10.20. Добавление действий в файле `ProductValuesController.cs` из папки `Controllers`

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

namespace SportsStore.Controllers {

  [Route("api/products")]
  public class ProductValuesController : Controller {
    private IWebServiceRepository repository;

    public ProductValuesController(IWebServiceRepository repo)
      => repository = repo;

    [HttpGet("{id}")]
    public object GetProduct(long id) {
      return repository.GetProduct(id) ?? NotFound();
    }

    [HttpGet]
    public object Products(int skip, int take) {
      return repository.GetProducts(skip, take);
    }

    [HttpPost]
    public long StoreProduct([FromBody] Product product) {
      return repository.StoreProduct(product);
    }

    [HttpPut]
    public void UpdateProduct([FromBody] Product product) {
      repository.UpdateProduct(product);
    }

    [HttpDelete("{id}")]
    public void DeleteProduct(long id) {
      repository.DeleteProduct(id);
    }
  }
}
```

Чтобы протестировать сохранение нового объекта, запустите приложение с применением `dotnet run` и выполните в новом окне PowerShell команду из листинга 10.21.

Листинг 10.21. Сохранение новых данных

```
Invoke-RestMethod http://localhost:5000/api/products
-Method POST -Body (@{Name="Scuba Mask"; Description="Spy on the Fish";
PurchasePrice=21; RetailPrice=40;CategoryId=1} |
ConvertTo-Json) -ContentType "application/json"
```

Набирать команду нелегко; она отправляет серверу HTTP-запрос `POST` со значениями для всех свойств, которые нужны для сохранения объекта `Product` в БД. После того как команда завершится, перейдите в окне браузера по ссылке `http://localhost:5000` и вы увидите новый объект (рис. 10.2).

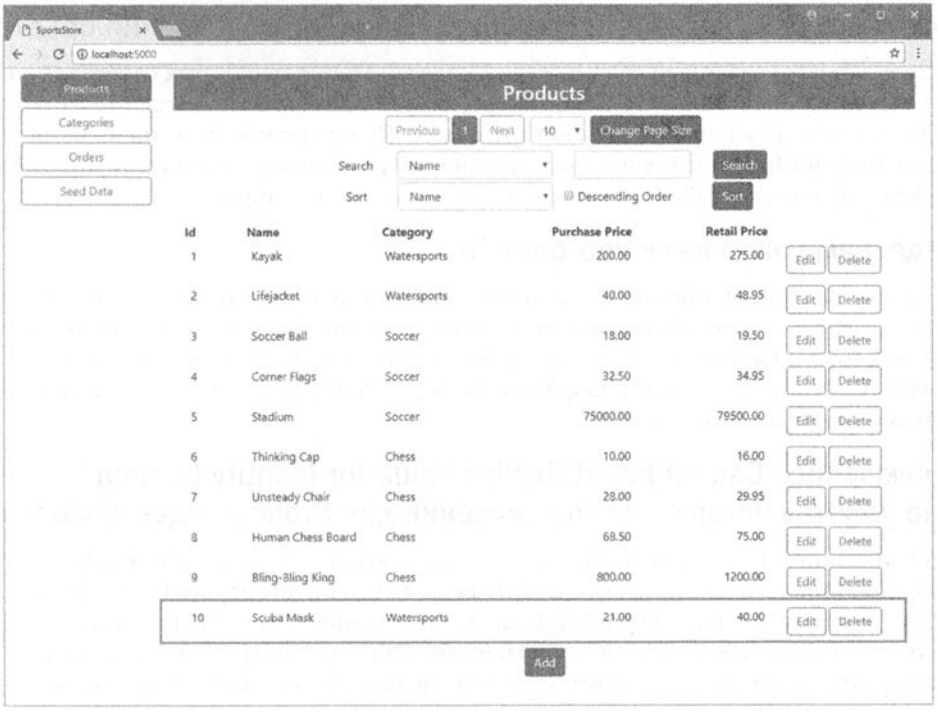

Рис. 10.2. Сохранение объекта через веб-службу REST

Для тестирования средства обновления выполните в окне PowerShell команду, приведенную в листинге 10.22, которая модифицирует товар `Kayak`.

Листинг 10.22. Модификация данных

```
Invoke-RestMethod http://localhost:5000/api/products -Method PUT -Body
(@{Id=1;Name="Green Kayak"; Description="A Tiny Boat"; PurchasePrice=200;
RetailPrice=300;CategoryId=1} | ConvertTo-Json) -ContentType "application/json"
```

Перезагрузите окно браузера; вы увидите, что значение `RetailPrice` для товара `Kayak` изменилось на `300`, а название теперь выглядит как `Green Kayak`.

Чтобы протестировать возможность удаления данных, выполните в окне PowerShell команду, показанную в листинге 10.23.

Листинг 10.23. Удаление объекта

```
Invoke-RestMethod http://localhost:5000/api/products/1 -Method DELETE
```

Распространенные проблемы и их решения

Большинство затруднений с веб-службами обусловлено проблемами сериализации, обсуждаемыми ранее в главе. Однако существуют и менее распространенные проблемы, которые описаны в последующих разделах.

Значения `null` у свойств при сохранении или обновлении объектов

Если объект, созданный связывателем моделей MVC, в некоторых свойствах содержит значения `null` или `0`, то наиболее вероятной причиной является отсутствие атрибута `FromBody` в параметре метода действия. По умолчанию для значений используется только URL, поэтому вы должны явно выбрать источник данных, если хотите, чтобы инфраструктура MVC задействовала другие части запроса.

Медленные запросы к веб-службе

Самой частой причиной медленных запросов является перечисление объекта `IQueryable<T>` и непредумышленное инициирование запроса. Это легко сделать, если вы обрабатываете данные перед сериализацией JSON, и важно помнить, что объект `IQueryable<T>` будет запрашивать БД всякий раз, когда он перечисляется, а не только в представлениях Razor.

Исключение "Cannot Insert Explicit Value for Identity Column" ("Не удается вставить явное значение для столбца идентичности")

Если вы получаете это исключение при написании веб-службы, то наиболее вероятная причина связана с тем, что клиент включил значение первичного ключа в объект, подлежащий сохранению в БД. В случае написания клиентского приложения самостоятельно вы можете принять меры, чтобы HTTP-запрос не включал значения для первичного ключа. Если ваша веб-служба используется сторонним приложением, то вы можете явно обнулить свойство первичного ключа в методе действия перед указанием инфраструктуре Entity Framework Core на необходимость сохранения данных.

Резюме

В главе работа над приложением `SportsStore` была завершена добавлением простой веб-службы REST, которая может применяться для снабжения данными клиентских приложений. Вы увидели распространенные проблемы, которые могут возникать при работе со связанными данными, и узнали, как их избежать, путем создания динамических типов. В следующей части книги начинается подробное описание функциональных средств Entity Framework Core.

ЧАСТЬ II

Подробные сведения об инфраструктуре Entity Framework Core 2

В второй части книги мы погрузимся в детали. Сначала будут исследоваться основные операции с данными, которые предлагает инфраструктура Entity Framework Core, и объясняться, каким образом управляются БД и как поддерживаются отношения между объектами. Также будут демонстрироваться различные способы использования Entity Framework Core с существующими данными.

ГЛАВА 11

Работа с Entity Framework Core

В этой главе будет показано, как инфраструктура Entity Framework Core применяется к проекту ASP.NET Core MVC, начиная с добавления пакетов NuGet и проработки создания элементарной модели данных, схемы БД и инфраструктуры для ее использования. Проект, созданный в настоящей главе, послужит фундаментом для последующих глав, в которых будут добавляться функциональные средства Entity Framework Core. В табл. 11.1 приведена сводка по главе.

Таблица 11.1. Сводка по главе

Задача	Решение	Листинг
Включение инструментов командной строки EF Core	Добавьте в файл `.csproj` элемент `DotNetCliToolReference`	11.10
Предоставление приложению доступа к функциональным средствам EF Core	Создайте класс контекста БД	11.11
Подготовка класса модели данных для хранилища в виде БД	Обеспечьте наличие конструкций `get` и `set` у всех свойств и добавьте свойство первичного ключа	11.12
Снабжение EF Core деталями о БД, которая должна применяться	Определите строку подключения	11.14
Подготовка БД для использования с EF Core	Создайте и примените миграцию БД	11.17
Гарантия обработки запросов в БД	Используйте интерфейс `IQueryable<T>`	11.24–11.26
Избегание дублированных запросов	Принудительно выполняйте запросы перед перечислением результатов	11.27, 11.28

Создание проекта ASP.NET Core MVC

В главе мы создадим проект с минимальным содержимым, требующимся для ASP.NET Core. Затем будут добавлены пакеты, классы и компоненты конфигурации, чтобы получить приложение MVC, которое использует Entity Framework Core.

Запустите Visual Studio, выберите в меню File (Файл) пункт New⇨Project (Создать⇨Проект) и укажите шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), чтобы создать проект по имени `DataApp` (рис. 11.1).

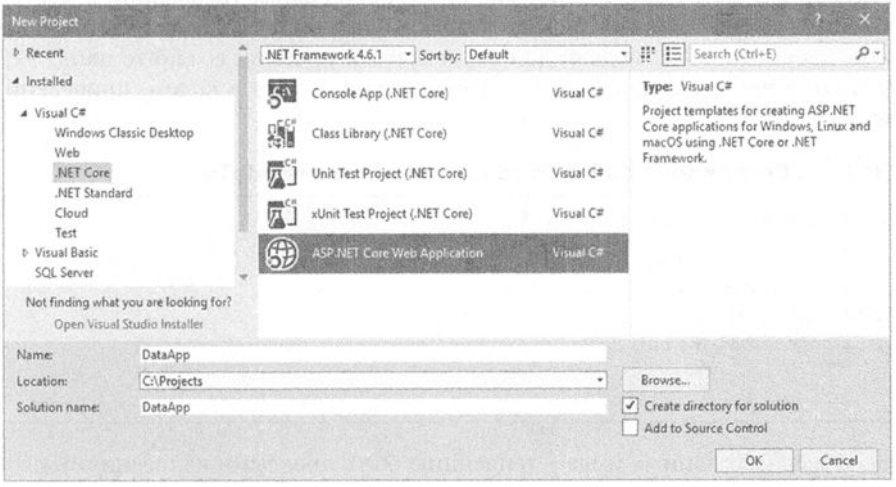

Рис. 11.1. Создание нового приложения

Щелкните на кнопке OK для перехода в новое диалоговое окно. Удостоверьтесь, что в списке в левой верхней части окна выбран вариант ASP.NET Core 2.0, и щелкните на шаблоне Empty (Пустой), как показано на рис. 11.2. Щелкните на кнопке OK, чтобы закрыть диалоговое окно и создать проект.

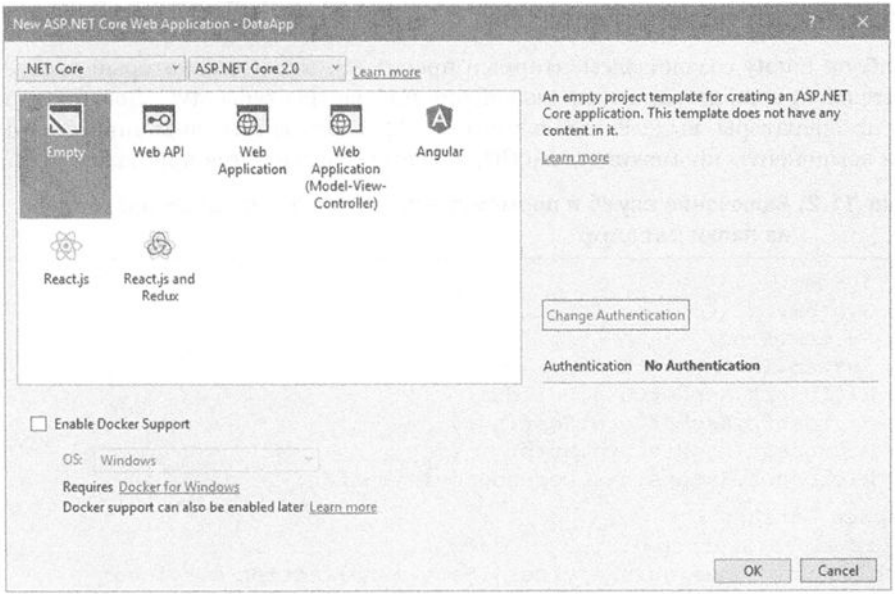

Рис. 11.2. Настройка проекта

Создание класса модели данных

Модель данных в приложении MVC определяется с применением обычных классов C#, по соглашению находящихся в папке под названием `Models`. Инфраструктура Entity Framework Core не выдвигает каких-то специальных требований к местоположению классов модели данных и будет благополучно работать с соглашением MVC.

Чтобы добавить класс модели данных в пример проекта, создайте папку `Models` и поместите в нее файл класса C# по имени `Product.cs` с кодом, приведенным в листинге 11.1.

Листинг 11.1. Содержимое файла `Product.cs` из папки `Models`

```
namespace DataApp.Models {

  public class Product {
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
  }
}
```

Классы моделей данных имеют тенденцию быть простыми коллекциями свойств, что облегчает работу с ними и гарантирует возможность их представления в формате наподобие JSON, который часто используется веб-службами HTTP, как демонстрировалось в главе 10. Класс `Product` в листинге 11.1 определяет свойства `Name`, `Category` и `Price` и является упрощенной моделью данных из примера `SportsStore`, который разрабатывался в первой части. В последующих главах сложность модели данных будет увеличена, но для начала вполне достаточно такого простого варианта.

Конфигурирование служб и промежуточного программного обеспечения

Шаблон Empty создает элементарный проект ASP.NET Core, который требует дополнительной конфигурации для включения инфраструктуры MVC. Добавьте в класс `Startup` операторы, выделенные в листинге 11.2, чтобы включить инфраструктуру MVC и компоненты промежуточного ПО, которые применяются при разработке.

Листинг 11.2. Включение служб и промежуточного ПО в файле `Startup.cs` из папки `DataApp`

```
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 DataApp {
  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();
    }
  }
}
```

Внесенные изменения делают доступными дружественные к разработчику сообщения об ошибках и поддержку статического содержимого (такого как файлы HTML и CSS files) и вдобавок настраивают инфраструктуру MVC со стандартной конфигурацией маршрутизации.

Добавление контроллера и представления

Теперь, когда инфраструктура MVC включена, можно создать контроллер и представление для обработки HTTP-запросов. Создайте папку `Controllers` и добавьте в нее файл класса C# по имени `HomeController.cs` с кодом из листинга 11.3.

Листинг 11.3. Содержимое файла `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {

    public IActionResult Index() {
      return View(new Product[] {
        new Product { Name = "P1", Category = "Cat1", Price = 10 },
        new Product { Name = "P2", Category = "Cat2", Price = 20 },
        new Product { Name = "P3", Category = "Cat3", Price = 30 },
      });
    }
  }
}
```

В контроллере имеется один метод действия под названием `Index()`, создающий коллекцию объектов-заполнителей `Product`, которые передаются как объекты модели представления стандартному представлению. Позже в главе, когда БД и Entity Framework окажутся в состоянии готовности к работе, эти статические объекты будут заменены.

Чтобы обеспечить согласованную компоновку для приложения, создайте папку по имени `Views/Shared` и поместите в нее страницу компоновки Razor под названием `_Layout.cshtml` с содержимым, приведенным в листинге 11.4.

Листинг 11.4. 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">
    <h4 class="bg-primary text-center p-2 text-white">@ViewData["Title"]</h4>
    @RenderBody()
  </div>
</body>
</html>
```

Чтобы снабдить метод действия `Index()` контроллера `Home` представлением, создайте папку `Views/Home` и добавьте в нее файл представления Razor по имени `Index.cshtml` с содержимым из листинга 11.5.

Листинг 11.5. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Product>
@{
  ViewData["Title"] = "Products";
  Layout = "_Layout";
}
<table class="table table-sm table-striped">
  <thead>
    <tr><th>Name</th><th>Category</th><th>Price</th></tr>
  </thead>
  <tbody>
    @foreach (var p in Model) {
      <tr>
        <td>@p.Name</td>
        <td>@p.Category</td>
        <td>$@p.Price.ToString("F2")</td>
      </tr>
    }
  </tbody>
</table>
```

Представление использует коллекцию объектов `Product`, получаемую от контроллера, в качестве своей модели представления для генерации строк в HTML-таблице, отображающих значения свойств `Name`, `Category` и `Price`. Чтобы включить вспомогательные функции дескрипторов, создайте в папке `Views` страницу импортирования представлений по имени `_ViewImports.cshtml` и добавьте операторы, показанные в листинге 11.6.

Листинг 11.6. Содержимое файла `_ViewImports.cshtml` из папки `Views`

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

Добавление инфраструктуры Bootstrap для стилизации CSS

Файл компоновки, созданный в листинге 11.4, включает элемент `link` для таблицы стилей CSS из инфраструктуры Bootstrap, с помощью которой стилизуется HTML-содержимое повсеместно в книге. Чтобы добавить Bootstrap в проект, щелкните правой кнопкой мыши на элементе проекта `DataApp` в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон элемента JSON File (Файл JSON), находящийся в категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), для создания файла по имени `.bowerrc` с содержимым, приведенным в листинге 11.7. (Важно обратить внимание на имя файла: оно начинается с точки, содержит две буквы `r` и не имеет расширения.)

Листинг 11.7. Содержимое файла `.bowerrc` из папки `DataApp`

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

Еще раз примените шаблон JSON File для создания файла по имени `bower.json` и поместите в него содержимое, приведенное в листинге 11.8.

Листинг 11.8. Содержимое файла `bower.json` из папки `DataApp`

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

Когда вы сохраните изменения, внесенные в файл, среда Visual Studio загрузит новую версию пакета Bootstrap и установит ее в папку `wwwroot/lib`.

Конфигурирование HTTP-порта

Изменение порта, который будет использоваться инфраструктурой ASP.NET Core для получения запросов, облегчит отслеживание примеров. Отредактируйте файл `launchSettings.json` из папки `Properties`, изменив URL, как продемонстрировано в листинге 11.9.

Листинг 11.9. Изменение HTTP-порта 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"
      }
    },
    "DataApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Указанные в файле URL применяются для конфигурирования приложения при его запуске с использованием IIS Express и в командной строке. Все примеры, рассматриваемые в книге, запускаются в командной строке, поэтому журнальные сообщения легко заметны.

Выполнение примера приложения

Применение инфраструктуры Entity Framework Core требует гораздо большей работы, чем может быть привычно для вас, а это значит, что запуск приложения в командной строке часто может оказываться самым естественным подходом, особенно учитывая легкость доступа к полезным журнальным сообщениям. Откройте окно командной строки или PowerShell, перейдите в папку проекта `DataApp` (ту, что содержит файл класса `Startup.cs`) и выполните команду из листинга 11.10.

Листинг 11.10. Выполнение примера приложения

```
dotnet run
```

Откройте новое окно браузера и перейдите по ссылке `http://localhost:5000`; вы увидите ответ, показанный на рис. 11.3. Удостоверившись в корректности работы приложения, остановите его с помощью комбинации клавиш <Ctrl+C>.

Добавление и конфигурирование инфраструктуры Entity Framework Core

Получив в свое распоряжение элементарное приложение ASP.NET Core MVC, самое время сконфигурировать инфраструктуру Entity Framework Core, чтобы она была в состоянии сохранять и извлекать данные по поручению приложения MVC.

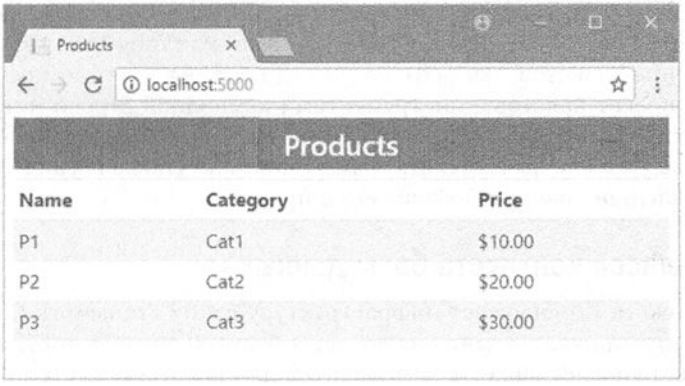

Рис. 11.3. Тестирование примера приложения

Добавление пакета NuGet

Созданный в Visual Studio проект сконфигурирован на использование мета-пакета ASP.NET Core, который появился в версии ASP.NET Core 2. Этот пакет NuGet предоставляет доступ ко всем индивидуальным пакетам, необходимым для разработки проекта, в том числе ASP.NET Core, ASP.NET Core MVC и Entity Framework Core, которые в ранних выпусках требовали добавления вручную длинного списка пакетов NuGet, прежде чем можно было начать процесс разработки.

Насколько бы ни был удобен мета-пакет, для проектов, в которых применяется инфраструктура Entity Framework Core, по-прежнему требуется одно добавление. Многие важные операции Entity Framework Core выполняются с использованием инструментов командной строки, а пакет NuGet, обеспечивающий эти инструменты, не входит в состав мета-пакета и должен устанавливаться вручную.

Добавьте пакет инструментов в файл `DataApp.csproj`, как показано в листинге 11.11, для чего щелкните правой кнопкой мыши на элементе проекта `DataApp` в окне Solution Explorer и выберите в контекстном меню пункт Edit DataApp.csproj (Редактировать `DataApp.csproj`).

Листинг 11.11. Добавление пакета инструментов в файле `DataApp.csproj` из папки `DataApp`

```
<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>
```

Пакет `Microsoft.EntityFrameworkCore.Tools.DotNet` добавляется с применением элемента `DotNetCliToolReference`, а не `PackageReference`, используемого для обычных пакетов. Элемент `DotNetCliToolReference` предназначен для добавления в проект пакетов с инструментами командной строки и не поддерживается стандартными инструментами NuGet, из-за чего файл должен быть добавлен вручную. После того как вы сохраните изменения, внесенные в файл `.csproj`, среда Visual Studio загрузит пакет и добавит его к проекту.

Создание класса контекста базы данных

Класс контекста БД снабжает инфраструктуру Entity Framework Core деталями о классах модели данных, экземпляры которых будут храниться в БД. Класс контекста — один из наиболее важных компонентов для работы с Entity Framework Core. Чтобы создать класс контекста, добавьте в папку `DataApp/Models` файл класса по имени `EFDatabaseContext.cs` с кодом, приведенным в листинге 11.12.

Листинг 11.12. Содержимое файла `EFDatabaseContext.cs` из папки `Models`

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

namespace DataApp.Models {
  public class EFDatabaseContext : DbContext {

    public EFDatabaseContext(DbContextOptions<EFDatabaseContext> opts)
      : base(opts) { }

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

Конструктор класса `EFDatabaseContext` принимает объект `DbContextOptions`, который применяется для конфигурирования Entity Framework Core. В случае приложений ASP.NET Core MVC конфигурирование производится в классе `Startup`, как демонстрируется далее в главе, а потому конструктор просто передает объект параметров базовому классу.

Совет. В классе контекста должен быть определен конструктор, даже если он не содержит какие-либо операторы. Попытка создать класс контекста без конструктора приведет к ошибке.

Классы контекста БД сообщают инфраструктуре Entity Framework Core о том, какие классы модели данных планируется хранить в БД, путем определения свойств, которые возвращают `DbSet<T>`, где параметр типа `T` указывает класс модели данных.

Подготовка сущностного класса

Инфраструктуре Entity Framework Core нужна возможность уникальной идентификации объектов `Product`, чтобы можно было сохранять их в БД и назначать им первичные ключи. По умолчанию Entity Framework Core ищет свойство по имени `Id` или

<Тип>Id для использования в качестве ключа, т.е. в случае класса Product свойство ключа может называться Id или ProductId. Чтобы подготовить класс Product для сохранения инфраструктурой Entity Framework Core, отредактируйте класс Product, добавив свойство ключа, как показано в листинге 11.13.

Листинг 11.13. Добавление свойства ключа в файле Product.cs из папки Models

```
namespace DataApp.Models {
  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
  }
}
```

Тип свойства ключа влияет на то, как оно будет обрабатываться инфраструктурой Entity Framework Core. Для числовых типов наподобие int и long схема будет настроена так, чтобы БД несла ответственность за генерирование уникальных значений ключа. Это удобно в ситуации, когда несколько экземпляров приложения ASP.NET Core MVC разделяют одну БД и координирование уникальных значений может быть сложным. В случае других типов, таких как string, свойство по-прежнему может применяться в качестве ключа, но за генерирование значений отвечает приложение. Расширенные возможности для работы с ключами рассматриваются в главе 19.

Обновление контроллера

Чтобы заменить статические данные объектами, получаемыми с использованием Entity Framework Core, откройте файл HomeController.cs и внесите изменения, представленные в листинге 11.14.

Листинг 11.14. Применение реальных данных в файле HomeController.cs из папки Controllers

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private EFDatabaseContext context;
    public HomeController(EFDatabaseContext ctx) {
      context = ctx;
    }
    public IActionResult Index() {
      return View(context.Products);
    }
  }
}
```

Конструктор класса контроллера будет получать объект контекста, который предоставит средство внедрения зависимостей ASP.NET Core, обеспечивающее связь между частями MVC и Entity Framework Core приложения.

Работа метода действия `Index()` заключается в снабжении своего представления коллекцией объектов `Product`, подлежащих отображению. Теперь коллекция получается за счет чтения свойства `Products` контекста БД, которое возвращает объект `DbSet<Product>`. Класс `DbSet<Product>` реализует интерфейс `IEnumerable<Product>`, т.е. представление `Index.cshtml` может перечислять последовательность объектов `Product` безо всяких изменений. (Однако, как объясняется в разделе "Избегание ловушки, связанной с использованием интерфейса `IEnumerable` вместо `IQueryable`" далее в главе, при работе с интерфейсом `IEnumerable` следует соблюдать осторожность.)

Конфигурирование поставщика базы данных

Инфраструктура Entity Framework Core не привязана к какому-то специфическому серверу баз данных. Взамен вся функциональность, требующаяся для специфического сервера баз данных, такого как MySQL или Microsoft SQL Server, содержится в пакете, который называется *поставщиком базы данных*. Это означает, что Entity Framework Core может применяться с любой БД, для которой доступен поставщик. Конфигурирование поставщика базы данных обычно выполняется за два шага: установка *строки подключения*, чтобы поставщик знал, каким образом взаимодействовать с сервером баз данных, и конфигурирование приложения, чтобы инфраструктуре Entity Framework Core было известно, какой поставщик БД должен использоваться для класса контекста.

Строка подключения предоставляет поставщику БД информацию, необходимую для подключения к специфическому серверу баз данных. В каждом поставщике применяется отличающийся формат строки подключения, но обычно в нем присутствует имя хоста и номер порта, требующиеся для сетевого подключения, пользовательские учетные данные, предназначенные для аутентификации, а также имя БД, которую нужно использовать. Во многих проектах применяются разные экземпляры сервера баз данных, так что данные действий, выполняемых разработчиками, хранятся отдельно от реальных данных пользователей, а потому в разное время будут требоваться разные строки подключения. Настройки конфигурации, которые изменяются часто, определяются в файле `appsettings.json`, так что их можно модифицировать без перекомпиляции проекта, и этот файл является обычным местом для помещения строк подключения в приложениях, использующих Entity Framework Core.

Определение строки подключения

Щелкните правой кнопкой мыши на элементе проекта `DataApp` в окне Solution Explorer, выберите в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент) и укажите шаблон элемента ASP.NET Configuration File (Файл конфигурации ASP.NET), чтобы создать файл по имени `appsettings.json` с содержимым из листинга 11.15.

Внимание! Строка подключения должна вводиться как одна неразрывная строка. На печатной странице представить ее так невозможно, из-за чего в листинге 11.15 применяется неуклюжее форматирование. В случае сомнений просмотрите содержимое файла `appsettings.json` в коде примеров для книги (`https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`).

Листинг 11.15. Содержимое файла `appsettings.json` из папки `DataApp`

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=DataAppDb;
      MultipleActiveResultSets=true"
  }
}
```

Строка подключения содержит детали, которые будут использоваться для подключения к серверу SQL Server. Ее части описаны в табл. 11.2.

На заметку! Эта строка подключения проще большинства других, поскольку в ней применяется версия сервера LocalDB, не требующая конфигурирования, которая устанавливается вместе с Visual Studio. Строки подключения для производственных серверов баз данных будут более сложными и обычно содержат имя хоста и номер TCP-порта плюс учетные данные, используемые при аутентификации.

Таблица 11.2. Элементы строки подключения к БД

Имя	Описание
`Server`	Эта настройка указывает имя хоста для сервера. В рассматриваемом примере она выглядит как `(localdb)\\MSSQLLocalDB`, потому что доступ к БД производится через LocalDB. Для БД других типов строка подключения, как правило, будет содержать имя хоста и номер TCP-порта, применяемые при подключении к серверу баз данных
`Database`	Эта настройка указывает имя БД. В текущем примере именем является `DataAppDb`
`MultipleActiveResultSets`	Эта настройка определяет, может ли клиент выполнять множество активных операторов SQL на одном подключении. Для приложений MVC она обычно устанавливается в `true`, поскольку в таком случае не возникают распространенные исключения при перечислении коллекций объектов данных в представлениях Razor

Конфигурирование приложения

Вы должны сконфигурировать приложение, чтобы инфраструктуре Entity Framework Core было известно, какой поставщик БД необходим, и какую строку подключения нужно использовать для доступа к БД. Конфигурирование производится в классе `Startup`, как показано в листинге 11.16.

Листинг 11.16. Конфигурирование поставщика БД в файле `Startup.cs` из папки `DataApp`

```
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 DataApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace DataApp {
  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<EFDatabaseContext>(options =>
        options.UseSqlServer(conString));
    }

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

При запуске приложения инфраструктура ASP.NET Core автоматически загружает содержимое файла `appsettings.json` и через интерфейс `IConfiguaton` открывает доступ к имеющимся в нем настройкам, в числе которых строки подключения. В листинге 11.16 конструктор получает объект реализации `IConfiguration` и присваивает его свойству по имени `Configuration`, которое затем применяется для конфигурирования Entity Framework Core в методе `ConfigureServices()`.

Конфигурирование инфраструктуры Entity Framework Core в методе `ConfigureServices()` начинается с получения строки подключения из данных конфигурации. Строка подключения получается с использованием наиболее прямолинейного подхода, при котором с помощью индексатора в стиле массивов указывается имя свойства конфигурации в одной строке. Сегменты строки представляют структуру файла `appsettings.json`, так что `ConnectionStrings:DefaultConnection` соответствует свойству `DefaultConnection` в разделе `ConnectionStrings` файла `appsettings.json`:

```
...
string conString = Configuration["ConnectionStrings:DefaultConnection"];
...
```

После получения строки подключения конфигурируется поставщик, который ассоциируется с классом контекста БД с применением расширяющего метода `AddDbContext()`. В параметре типа метода `AddDbContext()` указывается класс контекста и метод получает объект `DbContextOptionsBuilder`, который используется для выбора и конфигурирования поставщика БД:

```
...
services.AddDbContext<EFDatabaseContext>(
    options => options.UseSqlServer(conString));
...
```

Оператор идентифицирует класс `EFDatabaseContext` для Entity Framework Core. Метод `UseSqlServer()` выбирает поставщик БД для SQL Server и сообщает ему о необходимости подключения к БД с применением строки подключения, прочитанной из файла `appsettings.json`.

Конфигурирование ведения журналов Entity Framework Core

Многие примеры в книге основаны на понимании того, как операторы C# в приложении транслируются в SQL-запросы, отправляемые БД. Чтобы сконфигурировать систему ведения журналов ASP.NET Core для отображения полезных сообщений от инфраструктуры Entity Framework Core, внесите в файл `appsettings.json` добавления, приведенные в листинге 11.17.

Листинг 11.17. Включение ведения журналов EF Core в файле `appsettings.json` из папки `DataApp`

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

Система ведения журналов конфигурируется как часть процесса инициализации ASP.NET Core, выполняемого классом `Program`. Она читает данные конфигурации `Logging:LogLevel` и использует их для выбора, какие журнальные сообщения будут отображаться. Конфигурация ведения журналов в листинге 11.17 выбирает сообщения уровня `Information` из пространства имен `Microsoft.EntityFrameworkCore`, которые будут включать детали SQL-запросов, отправленных из приложения серверу баз данных. Журнальные сообщения из других пространств имен отключаются.

Генерирование и применение миграции

Для создания или модификации БД инфраструктура Entity Framework Core использует миграции, поэтому миграцию можно применить для сохранения данных приложения. Введите в окне командной строки или PowerShell команды из листинга 11.18, находясь внутри папки `DataApp`. Первая команда создает новую миграцию, которая содержит команды, создающие схему. Вторая команда применяет миграцию к БД.

Совет. Если в результате запуска команд из листинга 11.18 вы получаете сообщение "Сборка потерпела неудачу", то вероятная причина в том, что приложение в текущий момент выполняется. Остановите приложение, немного подождите и еще раз выполните команды для создания и применения миграции.

Листинг 11.18. Создание и применение миграции БД

```
dotnet ef migrations add Initial
dotnet ef database update
```

Работа миграций подробно объясняется в главе 13, а совокупным эффектом от этих команд будет создание БД по имени `DataAppDb` с таблицей `Products`, имеющей столбцы `Id`, `Name`, `Category` и `Price`, которые соответствуют свойствам, определенным в сущностном классе `Product`.

Конфигурация ведения журналов, созданная в листинге 11.17, гарантирует, что вы будете видеть SQL-команды, которые отправляются БД командой `dotnet ef database update`, применяющей миграцию. Самая важная часть показывает, как миграция создавала новую таблицу `Products`:

```
...
CREATE TABLE [Products] (
  [Id] bigint NOT NULL IDENTITY,
  [Category] nvarchar(max) NULL,
  [Name] nvarchar(max) NULL,
  [Price] decimal(18, 2) NOT NULL,
  CONSTRAINT [PK_Products] PRIMARY KEY ([Id])
);
...
```

Приведенная SQL-команда `CREATE TABLE` создает в таблице `Products` столбцы, которые соответствуют свойствам, определенным в классе модели `Product`. Именно так Entity Framework Core хранит данные приложения: объекты .NET хранятся как строки в таблице, а значение каждого свойства хранится в собственном столбце. Разумеется, инфраструктура Entity Framework Core делает намного большее, но памятуя об отображении объектов на строки таблиц, все остальное стает немного легче для понимания.

Миграция также создает таблицу по имени `__EFMigrationsHistory`, которую инфраструктура Entity Framework Core использует для отслеживания примененных миграций, но в рамках примера приложения таблица `Products` важнее, поскольку она будет хранить объекты `Product`.

Заполнение базы данных

Чтобы заполнить БД начальными данными, 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) и щелкните на меню Database Name (Имя базы данных), чтобы выбрать `DataAppDb` в списке, который отобразит имена баз данных, созданных с использованием LocalDB. Щелкните на кнопке Connect (Подключиться) и введите в редакторе содержимое листинга 11.19.

Листинг 11.19. Заполнение БД начальными данными

```
USE DataAppDb

INSERT INTO Products (Name, Category, Price) VALUES
```

```
( 'Kayak', 'Watersports', 275),
( 'Lifejacket', 'Watersports', 48.95),
( 'Soccer Ball', 'Soccer', 19.50),
( 'Corner Flags', 'Soccer', 34.95),
( 'Stadium', 'Soccer', 79500),
( 'Thinking Cap', 'Chess', 16),
( 'Unsteady Chair', 'Chess', 29.95),
( 'Human Chess Board', 'Chess', 75),
( 'Bling-Bling King', 'Chess', 1200)
SELECT Id, Name, Category, Price from Products
```

Введенные операторы добавляют строки в таблицу `Products` из БД `DataAppDb`, предоставляя значения для столбцов `Name`, `Category` и `Price`. Для столбца `Id` значения не указываются, т.к. они будут генерироваться БД, что обеспечит их уникальность.

Выберите пункт Execute (Выполнить) в меню SQL среды Visual Studio. Среда Visual Studio отправит SQL-операторы серверу SQL Server, где они будут выполнены и дадут результат, описанный в табл. 11.3.

Таблица 11.3. Начальные данные

Id	Name	Category	Price
1	Kayak (Каяк)	Watersports (Водный спорт)	275.00
2	Lifejacket (Спасательный жилет)	Watersports (Водный спорт)	48.95
3	Soccer Ball (Футбольный мяч)	Soccer (Футбол)	19.50
4	Corner Flags (Угловые флажки)	Soccer (Футбол)	34.95
5	Stadium (Стадион)	Soccer (Футбол)	79500.00
6	Thinking Cap (Мыслящая шапка)	Chess (Шахматы)	16.00
7	Unsteady Chair (Неустойчивый стул)	Chess (Шахматы)	29.95
8	Human Chess Board (Шахматная доска)	Chess (Шахматы)	75.00
9	Bling-Bling King (Блестящий король)	Chess (Шахматы)	1200.00

Запуск примера приложения

Все фрагменты на месте, так что самое время запустить приложение. В окне командной строки или PowerShell перейдите в папку проекта `DataApp` и выполните команду из листинга 11.20.

Листинг 11.20. Выполнение примера приложения

```
dotnet run
```

Приложение скомпилируется и запустится, а интегрированный веб-сервер ASP.NET Core начнет прослушивать HTTP-порт 5000 на предмет поступления запросов.

Откройте новое окно браузера и перейдите по ссылке `http://localhost:5000`; вы увидите, что инфраструктура Entity Framework Core снабдила приложение данными (рис. 11.4). Убедившись в корректном функционировании приложения, нажмите в командной строке комбинацию клавиш <Ctrl+C>, чтобы завершить работу.

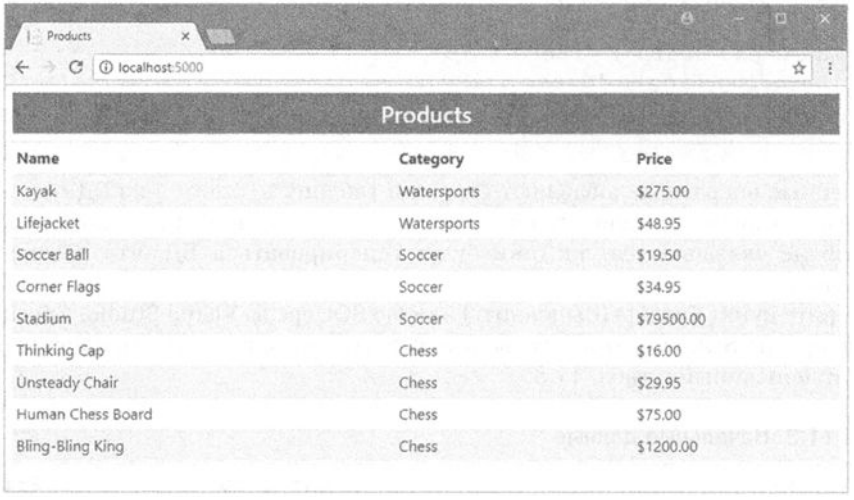

Рис. 11.4. Применение Entity Framework Core в примере приложения

Особенности работы приложения

Прежде чем двигаться дальше, стоит потратить некоторое время, чтобы понять, каким образом приложение получило данные, показанные на рис. 11.4.

Когда исполняющая среда ASP.NET Core получила от браузера HTTP-запрос на порте 5000, она направила его инфраструктуре MVC, которая с помощью своей системы маршрутизации выбрала для генерирования ответа действие `Index` контроллера `Home`.

Действие `Index` читает свойство `Products` объекта `EFDatabaseContext`, которое контроллер принимает через свой конструктор, и получает последовательность объектов `Product`. Эта последовательность передавалась в качестве модели представлению `Views/Home/Index.cshtml`, которое перечисляло содержащиеся в ней объекты для генерирования строк в HTML-таблице.

При перечислении последовательности объектов `Product` инфраструктура Entity Framework Core читает содержимое таблицы `Products` из БД `DataAppDb`, управляемой SQL Server посредством LocalDB. Имя БД и детали подключения указаны в строке подключения, определенной в файле `appsettings.json`, которая читается в классе `Startup`.

```
...
"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=DataAppDb;
 MultipleActiveResultSets=true"
...
```

Имя таблицы, содержащей данные, берется из имени, которое назначено свойству `DbSet<T>`, определенному в классе контекста `EFDatabaseContext`.

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

Инфраструктура Entity Framework Core использует это имя в миграции, применяемой для генерации схемы БД. Строки данных, прочитанные из таблицы, содержат значения столбцов `Id`, `Name`, `Category` и `Price` и используются для создания объектов `Product`, которые обрабатываются представлением `Index.cshtml`.

Изящество такого подхода связано с тем, насколько мало каждой части приложения приходится знать об остатке проекта, чтобы выполнять свою работу. Например, для работы с данными из БД представление `Index.cshtml` в изменении не нуждается; оно было создано для обработки `IEnumerable<Product>` до добавления к проекту инфраструктуры Entity Framework Core и по-прежнему используется. Инфраструктура Entity Framework Core позаботится о преобразовании строк из БД в объекты `Product` и сделает процесс работы с данными практически бесшовным. В контроллер потребуется внести ряд изменений, но как вы увидите в следующем разделе, их можно свести к минимуму, реализуя паттерн "Хранилище".

Если в будущих главах вы обнаружите, что запутались с тем, как работают определенные средства, тогда напомните себе этот пример и то, каким образом последовательность классов, настроек конфигурации и функций БД объединяются, чтобы снабдить данными часть MVC приложения. Пример поможет вспомнить, что инфраструктура Entity Framework Core не является магией, даже если иногда кажется такой. По мере продвижения в последующих главах вы увидите, как реализовано каждое средство, и что все строится на фундаменте, показанном в рассмотренном примере.

Реализация паттерна "Хранилище"

Для доступа к данным в БД контроллер `Home` в примере приложения работал напрямую с объектом `EFDatabaseContext`. Подход полностью функционален, но его можно улучшить за счет реализации паттерна "Хранилище".

Паттерн "Хранилище" состоит из интерфейса, определяющего операции над данными, которые могут выполняться в приложении, и класса реализации, делающего фактическую работу. Части MVC приложения пользуются только интерфейсом, в то время как "за кулисами" класс реализации выполняет операции над данными с применением контекста БД.

Использование хранилища облегчает изолирование компонентов MVC и объединяет код, взаимодействующий с инфраструктурой Entity Framework Core, что упрощает модульное тестирование различных частей приложения и переход на другую БД (или даже полную замену Entity Framework Core и применение другого уровня доступа к данным). В отсутствие хранилища код, который имеет дело с данными, распространяется по всему проекту, приводя к дублированию кода и затрудняя эффективное модульное тестирование.

Определение интерфейса и класса реализации хранилища

Отправной точкой для хранилища является определение интерфейса, который позволяет читать данные через инфраструктуру Entity Framework Core. Создайте в папке `Models` файл класса по имени `IDataRepository.cs` и поместите в него код, приведенный в листинге 11.21.

Внимание! Не реализуйте паттерн "Хранилище" в реальном проекте, пока не ознакомитесь с разделом "Избегание ловушки, связанной с использованием интерфейса `IEnumerable` вместо `IQueryable`" далее в главе. Код в листингах 11.21 и 11.22 содержит ловушку для неосторожных.

Листинг 11.21. Содержимое файла IDataRepository.cs из папки Models

```
using System.Collections.Generic;

namespace DataApp.Models {
  public interface IDataRepository {

    IEnumerable<Product> Products { get; }
  }
}
```

По мере добавления функциональных средств в будущих разделах интерфейс хранилища будет становиться более сложным, но в настоящий момент он определяет единственное свойство по имени Products, которое возвратит последовательность объектов Product. Создайте реализацию интерфейса хранилища, добавив в папку Models файл класса по имени EFDataRepository.cs с кодом из листинга 11.22.

Листинг 11.22. Содержимое файла EFDataRepository.cs из папки Models

```
using System.Collections.Generic;

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }

    public IEnumerable<Product> Products => context.Products;
  }
}
```

Класс реализации получает через свой конструктор объект EFDatabaseContext, который применяет для реализации свойства Products, требуемого интерфейсом IRepository. Это может не выглядеть как крупный шаг вперед, но означает, что контроллер Home может быть обновлен, чтобы зависеть от интерфейса хранилища, не нуждаясь в знании класса контекста (листинг 11.23).

Листинг 11.23. Использование хранилища в файле HomeController.cs из папки Controllers

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }
    public IActionResult Index() {
      return View(repository.Products);
    }
  }
}
```

Метод действия `Index()` работает таким же образом, как до введения хранилища, но получает коллекцию объектов `Product` через интерфейс хранилища. В итоге облегчается создание классов имитированных реализаций для модульного тестирования или построение реализаций, которые работают с другими уровнями доступа к данным.

Сконфигурируйте средство внедрения зависимостей для применения класса `EFDataRepository` в качестве реализации интерфейса `IRepository`, внеся в код класса `Startup` изменение из листинга 11.24.

Листинг 11.24. Конфигурирование внедрения зависимостей в файле `Startup.cs` из папки `DataApp`

```
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 DataApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace DataApp {
  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<EFDatabaseContext>(options =>
        options.UseSqlServer(conString));
      services.AddTransient<IDataRepository, EFDataRepository>();
    }

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

Снова запустите приложение, используя команду `dotnet run`; вы увидите те же самые данные. Отличие в том, что данные получаются через интерфейс хранилища, который устранил тесную связанность между частями MVC и EF Core приложения.

Избегание ловушки, связанной с использованием интерфейса IEnumerable вместо IQueryable

В способе реализации интерфейса хранилища из предыдущего раздела скрыта ловушка, в которую можно попасть по неосмотрительности. Чтобы выявить проблему, в приложение потребуется внести одно изменение, показанное в листинге 11.25, которое обновляет метод действия `Index()` контроллера `Home` с целью применения LINQ для фильтрации объектов, передаваемых представлению.

Листинг 11.25. Фильтрация объектов в файле HomeController.cs из папки Controllers

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      return View(repository.Products.Where(p => p.Price > 25));
    }
  }
}
```

Запрос LINQ выбирает только объекты `Product` со значением свойства `Price`, превышающим `25`. Запустите приложение, введя команду `dotnet run` в папке `DataApp`, и перейдите в браузере по ссылке `http://localhost:5000`. Вы увидите, что отображаются только объекты `Product`, значение свойства `Price` которых больше `25` (рис. 11.5).

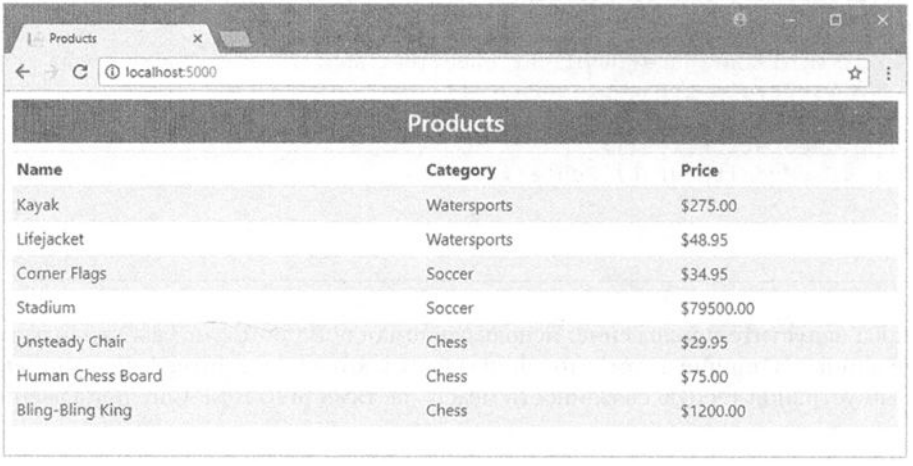

Рис. 11.5. Фильтрация данных

Если вы просмотрите журнальные сообщения в окне командной строки, то сможете заметить SQL-запрос, который инфраструктура Entity Framework Core использует для получения данных из БД:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
FROM [Products] AS [p]
...
```

Хотя требуются только некоторые строки таблицы `Products`, применяемый Entity Framework Core запрос не выполняет фильтрацию и извлекает из БД все данные `Product`. Такова индивидуальная особенность способа оценки LINQ своих запросов, т.е. все объекты `Product`, хранящиеся в БД, извлекаются в приложение, и лишь затем применяется фильтрация, указанная методом `Where()`.

В примере приложения это означает, что из таблицы `Products` в БД читаются все данные и используются для создания объектов `Product`, которые инспектируются LINQ-методом `Where()` и отбрасываются, если значение `Price` оказывается слишком малым.

В приложениях с небольшими объемами данных проблема несущественна. В рассматриваемом примере приложения в БД хранятся девять строк `Product`. Все девять строк читаются SQL-запросом и применяются для создания объектов `Product`, но только семь из них будут выбраны методом `Where()`. Для такого небольшого объема данных о факте извлечения инфраструктурой Entity Framework Core двух лишних строк с последующим их отбрасыванием контроллером MVC беспокоиться не стоит. Но в приложениях со значительными объемами данных, где из таблиц извлекается большое количество строк, используемых для создания объектов, которые далее просто отбрасываются, ситуация иная: это расточительная и затратная операция.

Чтобы исправить проблему, в интерфейс и класс реализации хранилища понадобится внести изменения. В листинге 11.26 приведены изменения интерфейса.

Листинг 11.26. Исправление проблемы с извлечением данных в файле IDataRepository.cs из папки Models

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

namespace DataApp.Models {
  public interface IDataRepository {

    IQueryable<Product> Products { get; }
  }
}
```

Интерфейс `IQueryable` унаследован от `IEnumerable`, но представляет запрос, который должен обрабатываться БД. В листинге 11.27 показаны соответствующие изменения, внесенные в класс реализации, так что он отражает новый возвращаемый тип для свойства `Products`.

Листинг 11.27. Исправление проблемы с извлечением данных в файле EFDataRepository.cs из папки Models

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

```
namespace DataApp.Models {
  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }
    public IQueryable<Product> Products => context.Products;
  }
}
```

Перезапустите приложение с помощью команды `dotnet run` и перейдите в браузере по ссылке `http://localhost:5000`. В журнальных сообщениях, выводимых в окне командной строки, вы увидите, что серверу SQL Server был оправлен другой запрос:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
FROM [Products] AS [p]
WHERE [p].[Price] > 25.0
...
```

Применение интерфейса `IQueryable` изменяет способ оценки запроса LINQ и обеспечивает выполнение фильтрации сервером баз данных, а не приложением MVC. Из БД извлекаются только данные, которые удовлетворяют критерию фильтра, а объекты не создаются лишь для того, чтобы немедленно быть отброшенными.

Осмысление и избегание проблемы с дополнительным запросом

Недостаток использования интерфейса `IQueryable<T>` связан с легкостью непредумышленной генерации большего числа запросов к БД, чем намечалось. Чтобы продемонстрировать проблему, модифицируйте метод действия в контроллере `Home` для выполнения множества операций над данными, полученными от Entity Framework Core (листинг 11.28).

Листинг 11.28. Выполнение множества операций над данными в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      var products = repository.Products.Where(p => p.Price > 25);
      ViewBag.ProductCount = products.Count();
      return View(products);
    }
  }
}
```

Новый оператор в методе Index() вызывает метод Count() на коллекции объектов, полученных из хранилища. Все выглядит достаточно безобидным, но если вы запустите приложение с применением dotnet run и перейдете по ссылке http://localhost:5000, то в журнальных сообщениях обнаружится, что БД были отправлены два запроса. Первый запрос предлагает серверу баз данных подсчитать количество товаров, соответствующих фильтру:

```
...
SELECT COUNT(*)
FROM [Products] AS [p]
WHERE [p].[Price] > 25.0
...
```

Второй запрос выполняет фактическое извлечение объектов данных:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
FROM [Products] AS [p]
WHERE [p].[Price] > 25.0
...
```

Интерфейс IQueryable<T> инициирует новый запрос при каждой оценке, т.е. один запрос отправляется, когда вызывается метод Count(), а второй запрос происходит во время перечисления представлением Razor объектов Product с целью заполнения HTML-таблицы, отправляемой браузеру. Запросы появляются в порядке, отличающемся от порядка следования операторов кода, поскольку представление Razor не визуализируется до тех пор, пока не будет завершен метод действия.

Проблема в том, что инфраструктура Entity Framework Core не обладает достаточным пониманием запросов LINQ, чтобы обнаружить возможность выполнения обеих операций над данными с использованием единственного запроса к БД. Избежать дополнительных запросов в такой ситуации можно, преобразовав объект реализации IQueryable<T> в обычный объект реализации IEnumerable<T> с применением методов ToArray() или ToList(), как показано в листинге 11.29.

Листинг 11.29. Избегание дополнительных запросов в файле HomeController.cs из папки Controllers

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      var products = repository.Products.Where(p => p.Price > 25).ToArray();
      ViewBag.ProductCount = products.Count();
      return View(products);
    }
  }
}
```

Метод `ToArray()` вызывает оценку запроса и производит коллекцию объектов, которые могут быть подвергнуты дальнейшей обработке, не инициируя дополнительные запросы, а это значит, что метод `Count()` оперирует на данных, уже извлеченных из БД, вместо выдачи нового запроса.

На заметку! Примеры в текущем разделе могут создать впечатление, что вы сталкиваетесь с постоянным риском либо запросить больше данных, либо отправить больше запросов, чем необходимо. Ключ в том, чтобы исследовать журнальные сообщения, сгенерированные из приложения, для проверки ожидаемого поведения, и это станет вашей второй натурой, когда вы привыкнете к работе с инфраструктурой Entity Framework Core.

Сокрытие операций над данными

Прямая проблема, связанная с использованием интерфейса `IQueryable<T>`, касается того, что он открывает остальным частям приложения детали управления данными. Таким образом, при написании метода действия, например, вы должны уделять внимание, применяется ли интерфейс `IQueryable<T>`, и удостоверяться в том, что запрашиваются только требующиеся данные с помощью только необходимого количества запросов.

Альтернативный и более удачный подход предусматривает сокрытие деталей запрашивания данных из хранилища, чтобы данными можно было пользоваться, не беспокоясь о том, как они были получены. В примере приложения это означает перенос кода, который выбирает данные, из контроллера в хранилище. В листинге 11.30 свойство `Products` интерфейса хранилища заменяется методом, выполняющим запрос, который ранее делался контроллером. Такой прием позволяет хранилищу возвращать объект реализации `IEnumerable<Product>`, что не допускает утечки деталей реализации, но также не приводит к извлечению и последующему отбрасыванию данных либо инициированию непредвиденных запросов.

Совет. Вы не обязаны скрывать операции над данными в своем приложении. Скрывая операции над данными, вы с меньшей вероятностью создадите непредвиденные запросы или запросите слишком много данных, но результат может оказаться менее гибким и привести к смещению сложности в хранилище. В своих проектах (и в примерах, рассмотренных в книге) я предпочитаю смешивать подходы и внимательно следить за журнальными сообщениями, генерируемыми приложением, чтобы гарантировать предсказуемый способ трансляции кода из части MVC приложения в SQL-запросы.

Листинг 11.30. Определение метода запроса в файле `IDataRepository.cs` из папки `Models`

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

namespace DataApp.Models {
  public interface IDataRepository {

    IEnumerable<Product> GetProductsByPrice(decimal minPrice);
  }
}
```

Здесь свойство Products было заменено методом GetProductsByPrice(). Часть MVC приложения будет применять этот метод вместо выполнения собственных запросов напрямую с использованием LINQ. В листинге 11.31 приведены изменения, которые потребуется внести в класс реализации хранилища.

Листинг 11.31. Определение метода запроса в файле EFDataRepository.cs из папки Models

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

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }
    public IEnumerable<Product> GetProductsByPrice(decimal minPrice) {
      return context.Products.Where(p => p.Price >= minPrice).ToArray();
    }
  }
}
```

Свойство DbSet<Product> класса контекста реализует интерфейс IQueryable <Product>. Таким образом, можно было бы просто возвратить результат из запроса LINQ без какого-либо приведения или преобразования, но был применен метод ToArray(), чтобы гарантировать, что будущее использование данных не вызовет выполнение дополнительных запросов.

В листинге 11.32 метод действия Index() контроллера Home обновляется с целью применения метода GetProductsByPrice(), оставляя работу по фильтрации данных хранилищу.

Листинг 11.32. Использование метода запроса данных в файле HomeController.cs из папки Controllers

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      var products = repository.GetProductsByPrice(25);
      ViewBag.ProductCount = products.Count();
      return View(products);
    }
  }
}
```

Метод действия способен получать интересующие его отфильтрованные данные, не беспокоясь о том, как они были получены, и может выполнять над ними дополнительные операции (например, LINQ-метод `Count()`) без необходимости учитывать, приведут ли они к нежелательному побочному эффекту.

Протестируйте внесенные изменения, запустив приложение посредством `dotnet run` и перейдя в браузере по ссылке `http://localhost:5000`. Объекты, отображаемые в браузере, не изменились. Однако хранилище больше не раскрывает детали реализации, что улучшает разделение между частями Entity Framework Core и MVC приложения, а также делает процесс работы с данными приложения не таким обременительным, хотя и менее гибким, поскольку контроллер может получать только специфический поднабор данных из БД.

Завершение примера приложения MVC

В заключение главы часть MVC приложения будет завершена за счет добавления методов действий и представлений для наиболее распространенных операций, требующихся приложению, а также поддерживающих их методов в хранилище. Реализация операций с применением Entity Framework Core рассматривается в следующей главе, а пока класс хранилища будет просто выводить сообщения в окно командной строки.

Завершение хранилища

Большинству приложений MVC нужны пять основных операций над данными: извлечение одиночного элемента, извлечение всех элементов, создание нового элемента, обновление существующего элемента и удаление элемента. Как только вы настроите и приведете в работоспособное состояние указанные пять операций, все остальное станет на свои места. Добавьте в интерфейс хранилища методы из листинга 11.33.

Листинг 11.33. Добавление методов в файле `IDataRepository.cs` из папки `Models`

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

namespace DataApp.Models {
  public interface IDataRepository {
    Product GetProduct(long id);
    IEnumerable<Product> GetAllProducts();
    void CreateProduct(Product newProduct);
    void UpdateProduct(Product changedProduct);
    void DeleteProduct(long id);
  }
}
```

В методах `GetProduct()` и `DeleteProduct()` определены параметры, которые принимают значение первичного ключа для сохраненного объекта, соответствующее свойству `Id`. Методы `CreateProduct()` и `UpdateProduct()` принимают объекты `Product`, а в методе `GetAllProducts()` параметры отсутствуют.

Добавьте в класс `EFDataRepository` методы из листинга 11.34 в качестве заполнителей для функциональности, которая будет реализована в последующих главах.

Исключением является метод GetAllProducts(), который просто возвращает значение свойства Products объекта контекста для обеспечения доступа ко всем объектам Product в БД.

Листинг 11.34. Добавление методов в файле EFDataRepository.cs из папки Models

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

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }

    public Product GetProduct(long id) {
      Console.WriteLine("GetProduct: " + id);
      return new Product();
    }

    public IEnumerable<Product> GetAllProducts() {
      Console.WriteLine("GetAllProducts");
      return context.Products;
    }

    public void CreateProduct(Product newProduct) {
      Console.WriteLine("CreateProduct: "
        + JsonConvert.SerializeObject(newProduct));
    }

    public void UpdateProduct(Product changedProduct) {
      Console.WriteLine("UpdateProduct : "
        + JsonConvert.SerializeObject(changedProduct));
    }

    public void DeleteProduct(long id) {
      Console.WriteLine("DeleteProduct: " + id);
    }
  }
}
```

Инфраструктура ASP.NET Core MVC полагается на пакет Newtonsoft.Json, который установлен как зависимость. В листинге 11.34 он используется для сериализации объектов, получаемых методами CreateProduct() и UpdateProduct(), так что объекты могут быть выведены на консоль и легко проанализированы.

Добавление методов действий

Чтобы добавить действия, которые будут работать с новыми операциями над данными, отредактируйте контроллер Home, добавив в него методы, как показано в листинге 11.35. В методах применяются соглашения и средства ASP.NET Core MVC для обработки запросов, включая методы только для запросов POST и привязку моделей.

Листинг 11.35. Добавление методов действий в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      return View(repository.GetAllProducts());
    }

    public IActionResult Create() {
      ViewBag.CreateMode = true;
      return View("Editor", new Product());
    }

    [HttpPost]
    public IActionResult Create(Product product) {
      repository.CreateProduct(product);
      return RedirectToAction(nameof(Index));
    }

    public IActionResult Edit(long id) {
      ViewBag.CreateMode = false;
      return View("Editor", repository.GetProduct(id));
    }

    [HttpPost]
    public IActionResult Edit(Product product) {
      repository.UpdateProduct(product);
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Delete(long id) {
      repository.DeleteProduct(id);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Обновление и добавление представлений

Для завершения части MVC приложения добавьте в папку `Views/Shared` новый файл представления Razor по имени `Editor.cshtml` и поместите в него разметку, приведенную в листинге 11.36. Представление `Editor` будет использоваться, когда пользователь пожелает отредактировать или создать элемент, и при изменении своего внешнего вида и действия, которому отправляется HTML-форма, полагается на свойство `ViewBag`, устанавливаемое методами `Create()` и `Edit()` в листинге 11.35.

Листинг 11.36. Содержимое файла `Editor.cshtml` из папки `Views/Shared`

```
@model DataApp.Models.Product
@{
  ViewData["Title"] = ViewBag.CreateMode ? "Create" : "Edit";
  Layout = "_Layout";
}

<form asp-action="@(ViewBag.CreateMode ? "Create" : "Edit")" method="post">
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </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>
```

Наконец, добавьте в представление `Index` кнопки, которые будут инициировать удаление элементов и начинать процессы создания и редактирования (листинг 11.37).

Листинг 11.37. Добавление кнопок в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<DataApp.Models.Product>
@{
  ViewData["Title"] = "Products";
  Layout = "_Layout";
}
<table class="table table-sm table-striped">
  <thead>
    <tr><th>ID</th><th>Name</th><th>Category</th><th>Price</th></tr>
  </thead>
  <tbody>
    @foreach (var p in Model) {
      <tr>
        <td>@p.Id</td>
        <td>@p.Name</td>
        <td>@p.Category</td>
        <td>$@p.Price.ToString("F2")</td>
        <td>
          <form asp-action="Delete" method="post">
            <a asp-action="Edit"
               class="btn btn-sm btn-warning" asp-route-id="@p.Id">
              Edit
            </a>
```

```
            <input type="hidden" name="id" value="@p.Id" />
            <button type="submit" class="btn btn-danger btn-sm">
              Delete
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create New Product</a>
```

Запустите приложение с применением команды `dotnet run` и перейдите по ссылке `http://localhost:5000`, чтобы увидеть результат, показанный на рис. 11.6. Часть MVC приложения завершена и паттерн "Хранилище" реализован, что обеспечивает фундамент для добавления функциональности в следующей главе.

Резюме

В главе было продемонстрировано добавление инфраструктуры Entity Framework Core в приложение ASP.NET Core MVC и различные способы реализации паттерна "Хранилище". Пример проекта будет расширяться, начиная со следующей главы, где вводятся основные операции над данными, которые поддерживает инфраструктура Entity Framework Core.

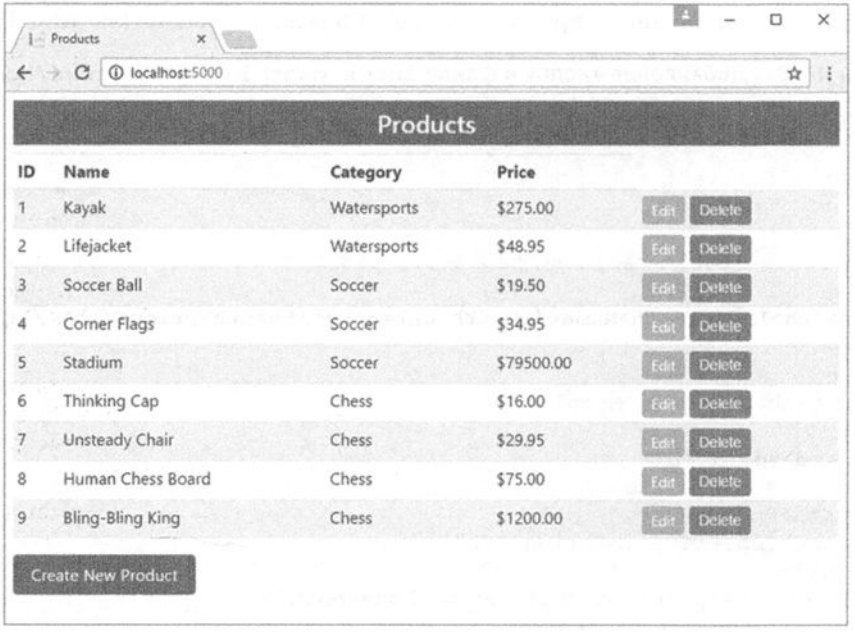

Рис. 11.6. Завершенная часть MVC приложения

ГЛАВА 12

Выполнение операций над данными

Модель данных в примере приложения слишком проста, чтобы считаться характерной для реальных проектов, однако она позволила объяснить, каким образом можно выполнять с использованием инфраструктуры Entity Framework Core четыре основных операции над данными (создание, чтение, обновление и удаление). В последующих главах будет показано, как создавать более сложные модели данных, но в текущей главе вы увидите, что даже простая модель способна выяснить очень многое о том, как работает Entity Framework Core.

В табл. 12.1 приведены сведения, позволяющие поместить основные операции над данными в контекст.

Таблица 12.1. Помещение основных операций над данными в контекст

Вопрос	Ответ
Что это такое?	Основные операции над данными позволяют читать, сохранять, обновлять и удалять данные из БД
Чем они полезны?	Операции являются фундаментальными строительными блоками при работе с инфраструктурой Entity Framework Core
Как они используются?	Класс контекста БД определяет свойство, которое предоставляет методы для сохранения, обновления и удаления данных и на котором можно выполнять запросы LINQ для чтения данных из БД
Существуют ли какие-то скрытые ловушки или ограничения?	Операции могут быть неэффективными и требовать от сервера выполнения большего объема работы, если только не предпринять шаги по применению функциональных средств наподобие обнаружения изменений
Существуют ли альтернативы?	Если вы используете Entity Framework Core, тогда такие операции сформируют фундамент для работы с данными

В табл. 12.2 приведена сводка по главе.

Таблица 12.2. Сводка по главе

Задача	Решение	Листинг
Получение из БД одиночного объекта	Применяйте метод `Find()`, который определяется объектом `DbSet<T>`, возвращенным классом контекста	12.5
Получение из БД всех объектов определенного типа	Выполните перечисление объекта `DbSet<T>`, который возвращается свойством класса контекста	12.6
Получение поднабора объектов	Используйте LINQ для выражения ограничений, которые обеспечат выбор требуемых объектов	12.7–12.10
Добавление объекта в БД	Передайте объект методу `DbSet<T>.Add()` и вызовите метод `SaveChanges()`	12.11, 12.12
Обновление объекта в БД	Передайте объект методу `DbSet<T>.Update()` и вызовите метод `SaveChanges()`	12.13
Применение средства обнаружения изменений для минимизации обновлений	Выполните запрос БД для получения исходного объекта, измените значения его свойств и вызовите метод `SaveChanges()`	12.14, 12.15
Использование средства обнаружения изменений без запрашивания исходных данных	Включите исходное состояние объекта в HTTP-запрос и применяйте его для предоставления исходных данных	12.16–12.19
Удаление объекта из БД	Передайте объект методу `DbSet<T>.Remove()` и вызовите метод `SaveChanges()`	12.21

Подготовительные шаги

В настоящей главе используется проект `DataApp`, созданный в главе 11. Чтобы провести подготовку, откройте окно командной строки или PowerShell, перейдите в папку проекта `DataApp` и введите команду из листинга 12.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 12.1. Удаление БД

```
dotnet ef database drop --force
```

Команда удаляет БД, что поможет вам получить ожидаемые результаты в последующих примерах. Выполните в папке проекта `DataApp` команду, показанную в листинге 12.2, чтобы заново создать БД.

Листинг 12.2. Подготовка БД

```
dotnet ef database update
```

Для заполнения БД начальными данными выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) и введите `(localdb)\MSSQLLocalDB` в поле Server Name (Имя сервера). Удостоверьтесь в том, что в поле Authentication (Аутентификация) выбран вариант Windows Authentication (Аутентификация Windows) и щелкните на меню Database Name (Имя базы данных), чтобы выбрать `DataAppDb` в списке, который отобразит имена баз данных, созданных с использованием LocalDB. Щелкните на кнопке Connect (Подключиться), чтобы открыть подключение к БД.

Совет. Вкладка хронологии поддерживает список баз данных, к которым вы подключались с применением Visual Studio, и позволяет подключаться к ним повторно, не предоставляя детали подключения.

Введите в редакторе SQL-команды из листинга 12.3. Выберите пункт Execute (Выполнить) в меню SQL среды Visual Studio. Среда Visual Studio предложит серверу баз данных выполнить SQL-команды, которые обеспечат наличие данных для работы с ними в приложении.

Листинг 12.3. Заполнение БД начальными данными

```
USE DataAppDb

INSERT INTO Products (Name, Category, Price)
VALUES
  ('Kayak', 'Watersports', 275),
  ('Lifejacket', 'Watersports', 48.95),
  ('Soccer Ball', 'Soccer', 19.50),
  ('Corner Flags', 'Soccer', 34.95),
  ('Stadium', 'Soccer', 79500),
  ('Thinking Cap', 'Chess', 16),
  ('Unsteady Chair', 'Chess', 29.95),
  ('Human Chess Board', 'Chess', 75),
  ('Bling-Bling King', 'Chess', 1200)
```

После выполнения SQL-команды вы увидите следующий результат, указывающий на то, в БД добавилось девять строк данных:

```
(9 row(s) affected)
(9 строк(а) затронуто)
```

Запуск примера приложения

Запустите приложение, выполнив команду из листинга 12.4 в папке проекта `DataApp`.

Листинг 12.4. Запуск на выполнение примера приложения

```
dotnet run
```

Откройте окно браузера и перейдите по ссылке `http://localhost:5000`. Приложение ASP.NET Core MVC будет использовать инфраструктуру Entity Framework для извлечения данных из БД и генерации ответа, представленного на рис. 12.1.

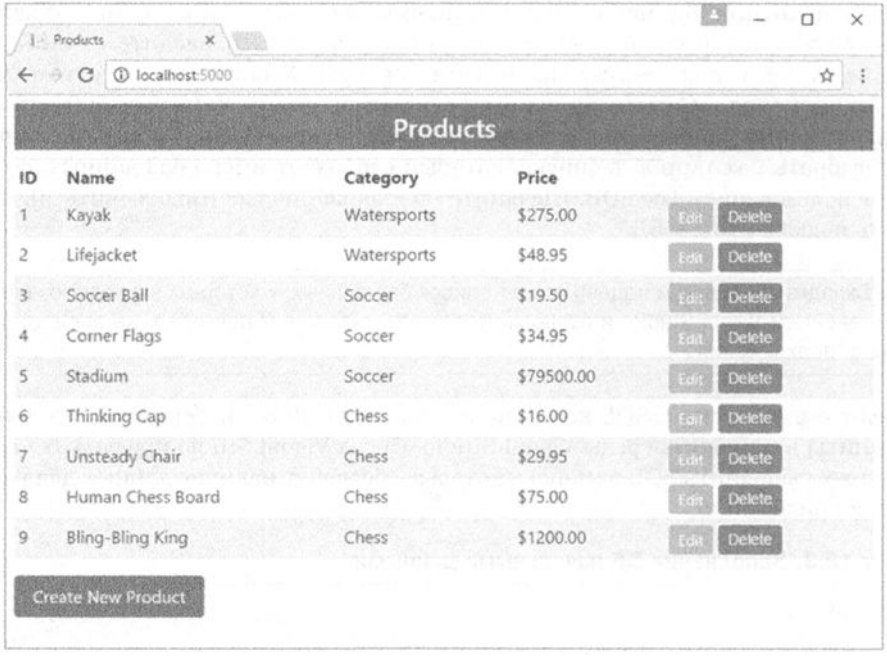

Рис. 12.1. Выполнение примера приложения

Чтение данных

Чтобы понять, как работает Entity Framework Core, лучше всего начать с запрашивания БД и извлечения данных, которые в ней содержатся. В последующих разделах будет объясняться, каким образом запросить одиночный сущностный объект, все объекты и только некоторые объекты.

Чтение объекта по ключу

Ключом к операциям над данными, применяемым в этой главе, является класс `DbSet<T>`, который используется в качестве результата для свойств, определяемых классом контекста БД. Вот определение свойства `Products` из класса `EFDatabaseContext`, определенного в главе 11:

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

Свойства вроде показанного выше исполняют две роли. Первая роль — сообщение инфраструктуре Entity Framework Core о том, что `Product` представляет собой сущностный класс, т.е. объекты `Product` будут храниться в БД. Эта информация важна при создании инфраструктурой Entity Framework Core миграции, поскольку она влияет на таблицы и строки, которые должны быть созданы для хранения данных в БД.

Вторая роль — предоставление свойства, которое позволяет операциям выполняться в БД, т.е. можно создавать, читать, обновлять и удалять объекты `Product` с применением Entity Framework Core. Класс `DbSet<T>` реализует интерфейсы и определяет методы, делающие такие операции возможными. Первый и самый элементарный метод называется `Find()` и для справки он описан в табл. 12.3.

Таблица 12.3. Метод класса DbSet<T> для запрашивания по ключу

Имя	Описание
Find(key)	Метод читает из таблицы строку, которая имеет указанный ключ, и возвращает представляющий ее объект. Если строки с таким ключом нет, то возвращается null. Если таблица требует составного ключа, как описано в главе 19, тогда его можно указать в виде множества параметров: Find(key1, key2, key3)

Метод Find() удобен, когда есть ключ и необходим ассоциированный с ним объект. В рамках примера приложения это означает наличие значения, присвоенного свойству Id объекта Product, который был сохранен в БД, и нужно извлечь полный объект Product.

Таково целевое назначение метода GetProduct() в хранилище, т.е. метод Find() может использоваться для реализации GetProduct(), как продемонстрировано в листинге 12.5.

Листинг 12.5. Запрашивание по ключу в файле EFDataRepository.cs из папки Models

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

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }

    public Product GetProduct(long id) {
      return context.Products.Find(id);
    }

    public IEnumerable<Product> GetAllProducts() {
      Console.WriteLine("GetAllProducts");
      return context.Products;
    }

    public void CreateProduct(Product newProduct) {
      Console.WriteLine("CreateProduct: "
        + JsonConvert.SerializeObject(newProduct));
    }

    public void UpdateProduct(Product changedProduct) {
      Console.WriteLine("UpdateProduct : "
        + JsonConvert.SerializeObject(changedProduct));
    }

    public void DeleteProduct(long id) {
      Console.WriteLine("DeleteProduct: " + id);
    }
  }
}
```

Метод `GetProduct()` применяется для снабжения объекта `Product` текущими значениями свойств, когда пользователь начинает процесс редактирования в части MVC приложения. Запустите приложение, используя `dotnet run`, перейдите в браузере по ссылке `http://localhost:5000` и щелкните на одной из кнопок Edit (Редактировать). Отобразится форма, заполненная данными, которые получены из объекта `Product`, созданного методом `Find()`, как показано на рис. 12.2.

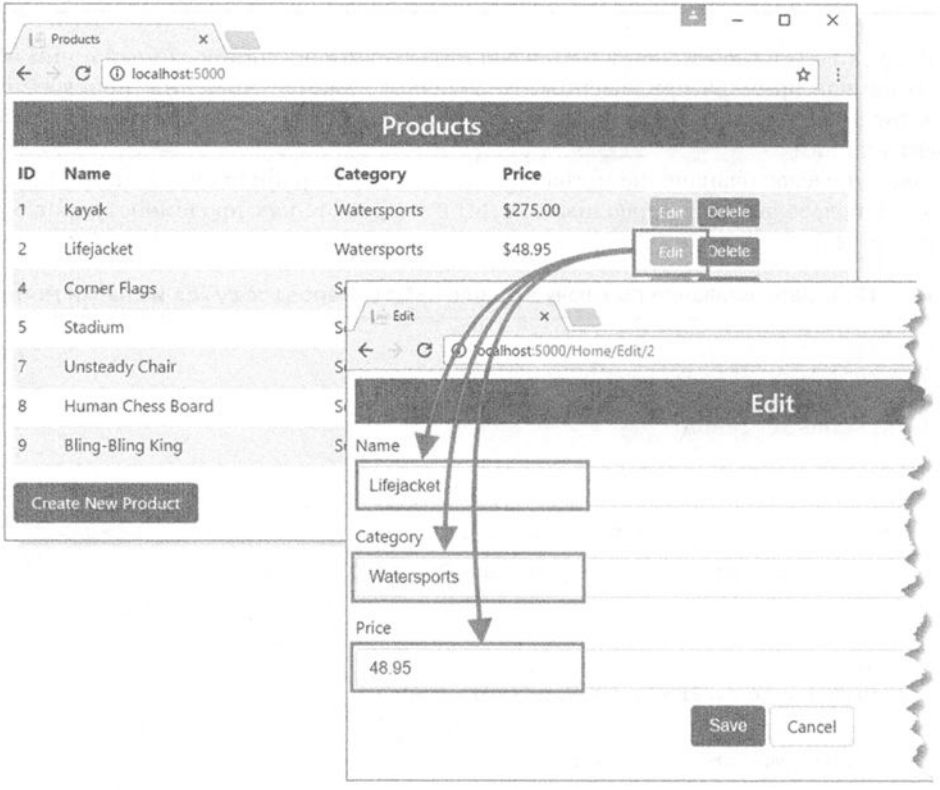

Рис. 12.2. Запрашивание одиночного объекта

Если вы просмотрите журнальные сообщения из приложения в окне командной строки, то заметите SQL-запрос, отправленный БД:

```
...
SELECT TOP(1) [e].[Id], [e].[Category], [e].[Name], [e].[Price]
FROM [Products] AS [e]
WHERE [e].[Id] = @__get_Item_0
...
```

Запрос извлекает значения для всех свойств, определенных классом `Product`, которые относятся к первой строке таблицы, имеющей указанное значение столбца `Id`. Полученные значения применяются методом `Find()` для создания объекта `Product`, который в итоге возвращается. Затем этот объект передается части MVC приложения и отображается пользователю.

Запрашивание всех объектов

Чтобы извлечь все объекты данных, хранящиеся в БД, можно прочитать значение свойства `Products` класса контекста, которое возвращает объект `DbSet<Product>`. Класс `DbSet<T>` реализует интерфейсы `IQueryable<T>` и `IEnumerable<T>`, т.е. с помощью цикла `foreach` можно выполнять перечисление последовательности объектов `Product`, прочитанных из БД.

Класс хранилища уже имеет поддержку для чтения всех объектов данных через метод `GetAllProducts()`, который был определен в главе 11:

```
...
public IEnumerable<Product> GetAllProducts() {
  return context.Products;
}
...
```

Инфраструктура Entity Framework Core не считывает данные из БД до тех пор, пока не начнется перечисление свойства `DbSet<T>`, что может стать источником путаницы. Чтобы продемонстрировать, каким образом Entity Framework Core откладывает считывание данных, приведите код действия `Index` контроллера `Home` в соответствие с листингом 12.6.

Листинг 12.6. Запрашивание всех объектов в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index() {
      var products = repository.GetAllProducts();
      System.Console.WriteLine("Property value has been read");
                                // Значение свойства прочитано
      return View(products);
    }

    public IActionResult Create() {
      ViewBag.CreateMode = true;
      return View("Editor", new Product());
    }

    // ...для краткости остальные действия не показаны...
  }
}
```

Оператор `System.Console.WriteLine()` в листинге 12.6 выводит сообщение, когда свойство `Products` было прочитано, но перед передачей значения методу `View()`.

Перезапустите приложения, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`. Просмотрите журнальные сообщения, сгенерированные приложением, и вы увидите, что оператор из листинга 12.6 выполнился перед SQL-запросом, который был отправлен БД:

```
...
Property value has been read
...
info: Microsoft.EntityFrameworkCore.Database.Sql[1]
      Executed DbCommand (6ms) [Parameters=[], CommandType='Text',
        CommandTimeout='30']
      SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
      FROM [Products] AS [p]
...
```

Данные извлекаются из БД, только когда на последовательности выполняется операция, такая как перечисление в цикле `foreach`, или когда применяется LINQ для преобразования последовательности в массив или список (с использованием метода `ToArray()` или `ToList()`).

Запрашивание специфических объектов

Свойства `DbSet<T>`, определяемые объектами контекста, также позволяют создавать более сложные запросы с применением средства LINQ to Entities. Класс `DbSet<T>` реализует интерфейс `IQueryable<T>`, который используется для создания запросов, выбирающих объекты из БД, как объяснялось в главе 11. Это делает возможным применение LINQ для запрашивания и обработки данных с тем преимуществом, что запрос выполняется сервером баз данных, а потому из БД считываются только совпадающие объекты.

В случае использования свойства `DBSet<T>` запрос не отправляется БД до тех пор, пока не начнется перечисление последовательности объектов. Таким образом, запросы могут строиться путем связывания в цепочки множества методов LINQ через несколько операторов кода, что хорошо вписывается в модель работы с приложениями MVC.

В качестве демонстрации модифицируйте файл представления `Index.cshtml`, добавив HTML-форму, которая позволит пользователю фильтровать список товаров, отображаемых в таблице (листинг 12.7).

Листинг 12.7. Добавление фильтрации в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<DataApp.Models.Product>
@{
  ViewData["Title"] = "Products";
  Layout = "_Layout";
}

<div class="m-1 p-2">
  <form asp-action="Index" method="get" class="form-inline">
    <label class="m-1">Category:</label>
    <select name="category" class="form-control">
      <option value="">All</option>
      <option selected="@(ViewBag.category == "Watersports")">
        Watersports
      </option>
      <option selected="@(ViewBag.category == "Soccer")">Soccer</option>
```

```
    <option selected="@(ViewBag.category == "Chess")">Chess</option>
  </select>
  <label class="m-1">Min Price:</label>
  <input class="form-control" name="price" value="@ViewBag.price" />
  <button class="btn btn-primary m-1">Filter</button>
 </form>
</div>
<table class="table table-sm table-striped">
 <thead>
  <tr><th>ID</th><th>Name</th><th>Category</th><th>Price</th></tr>
 </thead>
 <tbody>
  @foreach (var p in Model) {
   <tr>
    <td>@p.Id</td>
    <td>@p.Name</td>
    <td>@p.Category</td>
    <td>$@p.Price.ToString("F2")</td>
    <td>
     <form asp-action="Delete" method="post">
      <a asp-action="Edit" class="btn btn-sm btn-warning"
       asp-route-id="@p.Id">
       Edit
      </a>
      <input type="hidden" name="id" value="@p.Id" />
      <button type="submit" class="btn btn-danger btn-sm">
       Delete
      </button>
     </form>
    </td>
   </tr>
  }
 </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create New Product</a>
```

Пользователю предлагаются элемент `select` для выбора категории и элемент `input` для указания минимальной цены. Значения упомянутых элементов будут включены в запрос `GET`, отправляемый приложению, когда производится щелчок на кнопке Filter (Фильтровать).

Чтобы получить критерий фильтрации в приложении, отредактируйте действие `Index` контроллера `Home`, как показано в листинге 12.8.

Листинг 12.8. Получение критерия фильтрации в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
 public class HomeController : Controller {
  private IDataRepository repository;
```

```
    public HomeController(IDataRepository repo) {
      repository = repo;
    }
    public IActionResult Index(string category = null, decimal? price = null)
    {
      var products = repository.GetFilteredProducts(category, price);
      ViewBag.category = category;
      ViewBag.price = price;
      return View(products);
    }
    public IActionResult Create() {
      ViewBag.CreateMode = true;
      return View("Editor", new Product());
    }
    // ...для краткости другие действия не показаны...
  }
}
```

Метод действия `Index()` определяет два необязательных параметра, которые передаются методу хранилища по имени `GetFilteredProducts()`. Для создания метода `GetFilteredProducts()`, применяемого действием `Index`, расширьте интерфейс хранилища согласно листингу 12.9.

Совет. В листинге 12.8 для параметра `price` использовалось значение типа `decimal`, допускающего `null`, чтобы провести различие между ситуациями, когда пользователь ничего не вводит (и тогда параметр будет равен `null`) и когда пользователь вводит ноль (и тогда параметр будет равен `0`). Применение обычного параметра `decimal` привело бы в результате к нулевому значению в обеих ситуациях.

Листинг 12.9. Добавление метода в файле `IDataRepository.cs` из папки `Models`

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

namespace DataApp.Models {
  public interface IDataRepository {

    Product GetProduct(long id);

    IEnumerable<Product> GetAllProducts();

    IEnumerable<Product> GetFilteredProducts(string category = null,
      decimal? price = null);

    void CreateProduct(Product newProduct);

    void UpdateProduct(Product changedProduct);

    void DeleteProduct(long id);
  }
}
```

Финальный шаг предусматривает реализацию метода для построения запроса LINQ на основе значений, которые предоставил пользователь. Модифицируйте класс хранилища, реализовав метод `GetFilteredProducts()`, как показано в листинге 12.10.

Листинг 12.10. Построение запроса в файле EFDataRepository.cs из папки Models

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

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }

    public Product GetProduct(long id) {
      return context.Products.Find(id);
    }

    public IEnumerable<Product> GetAllProducts() {
      Console.WriteLine("GetAllProducts");
      return context.Products;
    }

    public IEnumerable<Product> GetFilteredProducts(string category = null,
        decimal? price = null) {
      IQueryable<Product> data = context.Products;
      if (category != null) {
        data = data.Where(p => p.Category == category);
      }
      if (price != null) {
        data = data.Where(p => p.Price >= price);
      }
      return data;
    }

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

Реализация метода `GetFilteredProducts()` начинается чтением значения свойства `Product` класса контекста и его присваиванием переменной типа `IQueryable<Product>`. Как объяснялось в главе 11, использование интерфейса `IQueryable<Product>` гарантирует, что фильтрация данных выполняется в БД вместо загрузки всех объектов с их последующим фильтрованием.

Запрос строится на основе того, были ли получены значения для параметров `category` и `price`. При наличии значений параметров значение переменной типа `IQuerable<Product>` обновляется с применением LINQ-метода `Where()`. Запрос к БД не выдается до тех пор, пока в представлении не начнется перечисление объекта реализации `IQueryable<T>`, что позволяет использовать методы LINQ для избирательного составления запроса по нескольким строкам кода.

Чтобы увидеть эффект от внесенных изменений, перезапустите приложение, перейдите по ссылке `http://localhost:5000` и введите в полях формы значения, приведенные на рис. 12.3, для фильтрации данных, отображаемых приложением.

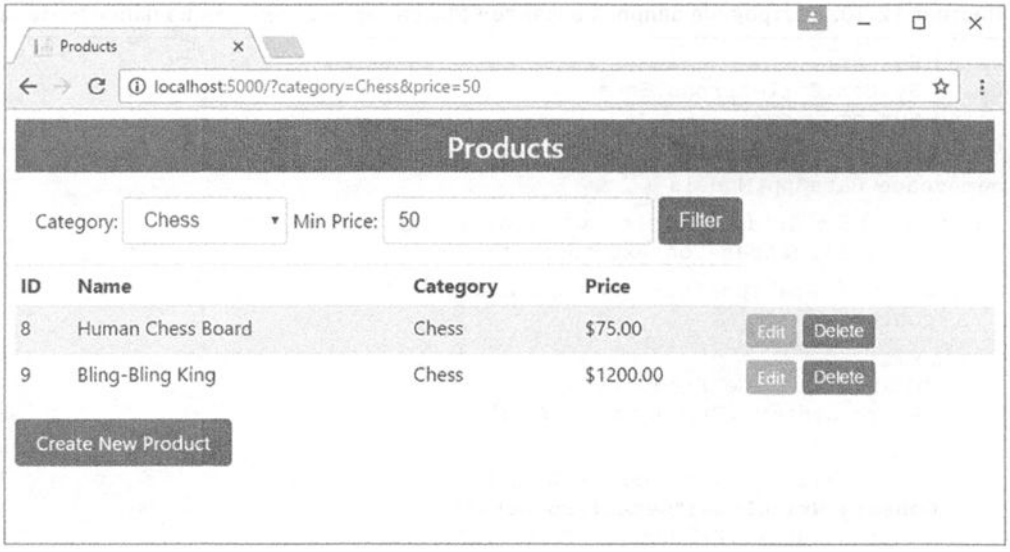

Рис. 12.3. Фильтрация данных в примере приложения

Возможны четыре типа запросов:

- запрашивание всех объектов, когда оба параметра оказываются `null`;
- запрашивание объектов в специфической категории без указания минимальной цены;
- запрашивание объектов с минимальной ценой во всех категориях;
- запрашивание объектов в специфической категории с указанием минимальной цены.

Вы можете просмотреть каждый запрос, который будет отправляться серверу БД, выбирая значения категории и вводя числа в поле цены. Например, вот запрос, который отправится, если вы выберете категорию, но не введете минимальную цену:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
FROM [Products] AS [p]
WHERE [p].[Category] = @__category_0
...
```

А такой запрос применяется в случае выбора категории и ввода минимальной цены:

```
...
SELECT [p].[Id], [p].[Category], [p].[Name], [p].[Price]
FROM [Products] AS [p]
WHERE ([p].[Category] = @__category_0) AND ([p].[Price] >= @__price_1)
...
```

Ключевой момент в том, что выбор данных выполняется с помощью запроса, который отправляется БД, обеспечивая возвращение приложению только совпадающих данных.

Сохранение новых данных

Следующим шагом будет добавление возможности сохранения новых объектов в БД. Отредактируйте класс хранилища для реализации метода `CreateProduct()`, как иллюстрируется в листинге 12.11.

Листинг 12.11. Сохранение данных в файле `EFDataRepository.cs` из папки `Models`

```
...
public void CreateProduct(Product newProduct) {
  newProduct.Id = 0;
  context.Products.Add(newProduct);
  context.SaveChanges();
}
...
```

В примере приложения есть два источника объектов `Product`: процесс привязки модели MVC и объект контекста БД. Процесс привязки модели создает объекты `Product`, когда он получает HTTP-запрос `POST`, а объект контекста БД создает объекты `Product`, когда он читает данные из БД.

Инфраструктура Entity Framework Core несет ответственность за объекты `Product`, создаваемые объектом контекста БД, но не видит те, которые создаются связывателем моделей MVC. Метод `Add()`, вызываемый на свойстве `DbSet<T>` контекста, сообщает инфраструктуре Entity Framework Core об объекте `Product`, который был создан где-то в другом месте приложения, так что он может быть записан в БД.

Совет. Свойство `DbSet<T>` также определяет метод `AddRange()`, который можно использовать для сохранения множества объектов в одном вызове, как объясняется в главе 13.

Метод `SaveChanges()` сохраняет в БД ожидающие изменения, внесенные в объекты `Product`, которые управляются Entity Framework Core. Сюда входят любые объекты `Product`, переданные методу `Add()`. Результатом выполнения кода в листинге 12.11 будет уведомление инфраструктуры Entity Framework Core об объекте `Product` и его сохранение в БД.

Чтобы увидеть эффект, перезапустите приложение с применением `dotnet run`, перейдите в браузере по ссылке `http://locahost:5000` и щелкните на кнопке Create (Создать). Заполните поля формы и щелкните на кнопке Save (Сохранить).

Когда вызывается метод действия `Create()` контроллера `Home` для обработки HTTP-запроса `POST`, он получает в качестве своего параметра объект `Product`, созданный связывателем моделей. Метод действия `Create()` обращается к методу `CreateProduct()` хранилища, который с помощью метода `Add()` сообщает инфраструктуре Entity Framework Core об объекте `Product` и посредством метода `SaveChanges()` сохраняет его в БД. Новые данные отобразятся в таблице товаров (рис. 12.4).

Назначение ключей

Обратите внимание, что метод `CreateProduct()` явно устанавливает в ноль значение свойства `Id` объекта `Product`:

```
...
newProduct.Id = 0;
...
```

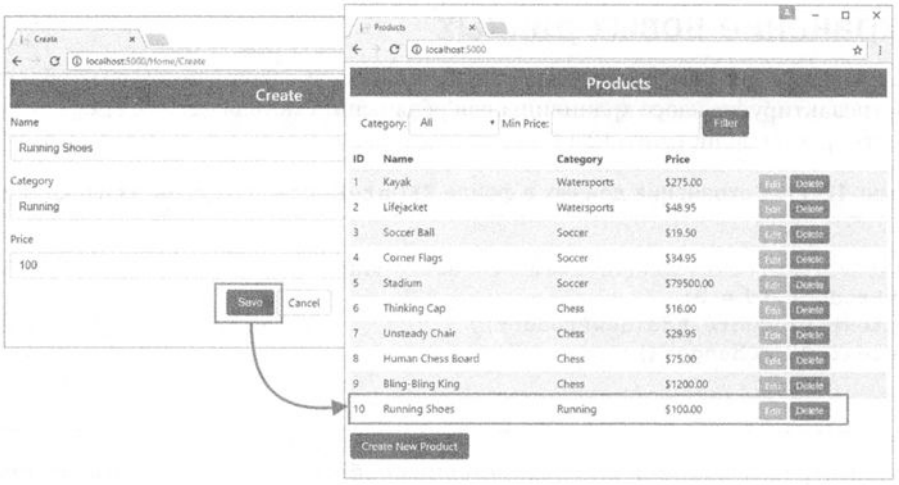

Рис. 12.4. Сохранение новых данных

Значения первичного ключа для новых объектов назначаются сервером баз данных при создании новых строк в таблице, и в случае ненулевого значения свойства `Id` сгенерируется исключение. В листинге 12.11 свойство `Id` явно устанавливается в ноль; тем самым гарантируется, что значение, полученное из HTTP-запроса, использоваться не будет.

На заметку! Можно было бы сконфигурировать связыватель моделей MVC так, чтобы он игнорировал свойство `Id` в HTTP-запросе, но это предполагает, что связыватель моделей является единственным источником объектов `Product`, передаваемых методу `CreateProduct()`. В листинге 12.11 преследовалась цель, чтобы часть Entity Framework Core приложения была надежна сама по себе, а потому свойство `Id` явно устанавливалось в ноль.

Когда инфраструктура Entity Framework Core сохраняет новый объект, она немедленно выполняет SQL-запрос для выяснения значения, которое сервер баз данных присвоил столбцу `Id` в новой строке таблицы. Вы можете увидеть его среди SQL-команд, находящихся в журнальных сообщениях, которые сгенерировало приложение при создании нового товара:

```
...
INSERT INTO [Products] ([Category], [Name], [Price])
VALUES (@p0, @p1, @p2);
SELECT [Id]
FROM [Products]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

Оператор `INSERT` сообщает серверу баз данных о необходимости создания новой строки в таблице `Products` и предоставляет значения для столбцов `Category`, `Name` и `Price`. Оператор `SELECT` запрашивает значение столбца `Id`, которое применялось при создании новой строки. Возвращаемое значение используется для обновления объекта `Product`, обеспечивая его согласованность с представлением в БД.

Просмотреть значение ключа можно, прочитав свойство `Id` после вызова метода `SaveChanges()`, как показано в листинге 12.12.

Листинг 12.12. Установление значения ключа в файле EFDataRepository.cs из папки Models

```
...
public void CreateProduct(Product newProduct) {
  newProduct.Id = 0;
  context.Products.Add(newProduct);
  context.SaveChanges();
  Console.WriteLine($"New Key: {newProduct.Id}");
                    // Новый ключ:
}
...
```

Перезапустите приложение с применением `dotnet run` и повторите процесс создания нового элемента данных. В выводе приложения вы увидите сообщение, которое отображает значение ключа, назначенное новому объекту `Product`:

```
New Key: 11
Новый ключ: 11
```

Значение ключа также присутствует в таблице товаров, отображаемой браузером.

Обновление данных

Процесс модификации существующих данных похож на процесс сохранения новых данных, но требует небольшой работы по обеспечению его эффективности. Есть три подхода к обновлению, которые описаны в последующих разделах.

Обновление полного объекта

Простейший подход к выполнению обновления предусматривает передачу объекта `Product`, созданного процессом привязки модели MVC, под управление инфраструктуры Entity Framework Core подобно процессу сохранения новых данных. Чтобы добавить поддержку такого обновления данных, приведите метод `UpdateProduct()` класса `EFDataRepository` в соответствие с листингом 12.13.

Листинг 12.13. Обновление данных в файле EFDataRepository.cs из папки Models

```
...
public void UpdateProduct(Product changedProduct) {
  context.Products.Update(changedProduct);
  context.SaveChanges();
}
...
```

Метод `DbSet<T>.Update()` используется для сообщения инфраструктуре Entity Framework Core о том, что объект `Product` был модифицирован, а метод `SaveChanges()` контекста записывает объект в БД.

Совет. Свойство `DbSet<T>` также определяет метод `UpdateRange()`, который может применяться для обновления множества объектов в одном вызове, как объясняется в главе 15.

Перезапустите приложение, используя команду `dotnet run`, щелкните на кнопке Edit (Редактировать) для товара `Stadium` и измените значение в поле Name (Наименование) на `Stadium (Large)`. Щелкните на кнопке Save (Сохранить); в результате БД обновится, а изменения отразятся в списке товаров, когда браузер будет перенаправлен на действие `Index` (рис. 12.5).

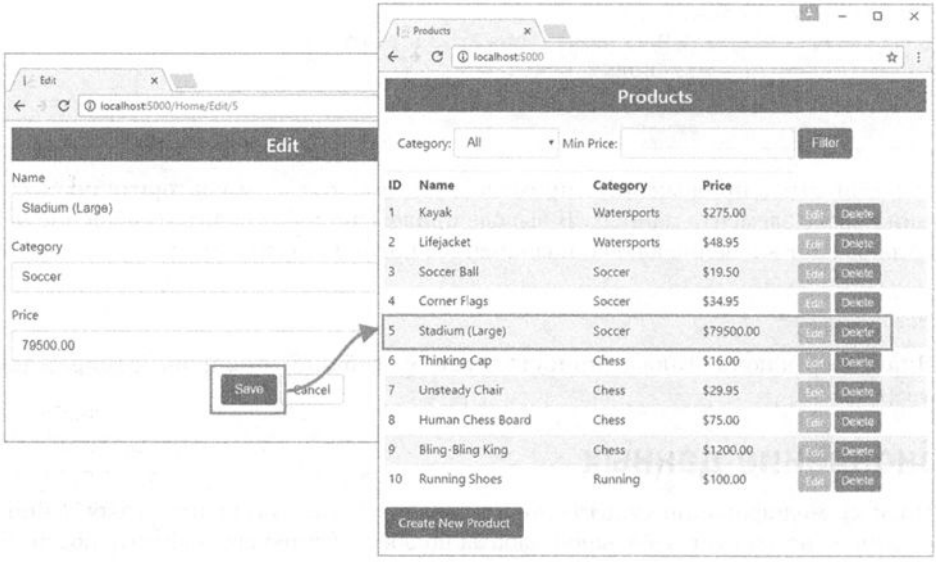

Рис. 12.5. Обновление объекта

Преимущество этого подхода связано с простотой: для обработки обновлений требуется всего две строки кода. Его недостаток в том, что инфраструктура Entity Framework Core знает об изменении объекта `Product`, но не располагает достаточной информацией о том, что было изменено только одно свойство. Таким образом, в БД должны сохраняться все свойства объекта `Product` и потому SQL-команда обновляет три имеющихся свойства:

```
...
UPDATE [Products] SET [Category] = @p0, [Name] = @p1, [Price] = @p2
WHERE [Id] = @p3;
...
```

Хотя изменилось только одно свойство, в БД обновляются все три свойства. Для простых объектов запись неизмененных свойств не будет значительной проблемой, но в реальных проектах со сложными моделями данных такой подход может оказаться неэффективным.

Совет. Если вы просмотрите консольный вывод приложения, то заметите, что оператор `UPDATE` находится в середине трех операторов, отправляемых серверу баз данных. Первый оператор, `SET NOCOUNT ON`, с целью повышения производительности отключает средство, сообщающее количество затронутых строк. Второй оператор, `UPDATE`, обновляет модифицированную строку или строки в таблице. Третий оператор, `SELECT @@ROWCOUNT`, сообщает о том, сколько строк было затронуто оператором `UPDATE`. Инфраструктура Entity Framework Core всегда проверяет, что было изменено ожидаемое количество строк.

Запрашивание существующих данных перед обновлением

Инфраструктура Entity Framework Core умеет точно определять, какие свойства объекта были модифицированы, чем можно воспользоваться и избежать записывания неизмененных данных в БД. Чтобы выяснить, каким образом Entity Framework Core обнаруживает изменения, обновите метод `UpdateProduct()` хранилища, как показано в листинге 12.14.

Листинг 12.14. Применение средства обнаружения изменений в файле `EFDataRepository.cs` из папки `Models`

```
...
public void UpdateProduct(Product changedProduct) {
  Product originalProduct = context.Products.Find(changedProduct.Id);
  originalProduct.Name = changedProduct.Name;
  originalProduct.Category = changedProduct.Category;
  originalProduct.Price = changedProduct.Price;
  context.SaveChanges();
}
...
```

В приведенном примере присутствуют два объекта `Product`. Объект `changedProduct` получается в параметре метода действия и был создан связывателем моделей MVC с использованием данных HTTP-запроса `POST`. Объект `originalProduct` создавался инфраструктурой Entity Framework Core как результат вызова метода `Find()` и представляет данные, которые в текущий момент находятся в БД.

Инфраструктура Entity Framework Core отслеживает создаваемые ею объекты с применением данных из БД и определяет, когда значения свойств изменились. Чтобы задействовать такое средство в своих интересах, значения свойств объекта `changedProduct` присваиваются свойствам объекта `originalProduct`, после чего вызывается метод `SaveChanges()`. Инфраструктура Entity Framework Core проинспектирует значения свойств объекта `originalProduct` с целью выяснения, изменились ли они с момента создания, и обновит только те свойства, значения которых стали отличаться.

Перезапустите приложение, снова щелкните на кнопке Edit для товара `Stadium` и измените значение `Stadium (Large)` в поле Name на `Stadium (Small)`. После щелчка на кнопке Save журнальные сообщения покажут SQL-оператор, который обновляет только измененное свойство:

```
...
UPDATE [Products] SET [Name] = @p0
WHERE [Id] = @p1;
...
```

Процесс обнаружения изменений идентифицирует, что свойство `Name` изменилось, и серверу баз данных отправляется оператор, который обновит только это свойство.

Процесс обнаружения изменений

В базовом классе контекстов БД, `DbContext`, определен метод `Entry()`, возвращающий объект `EntityEntry`, который инфраструктура Entity Framework Core использует для обнаружения изменений в создаваемых ею объектах. Наиболее полезные свойства `EntityEntry` описаны в табл. 12.4.

Таблица 12.4. Полезные свойства `EntityEntry`

Имя	Описание
`State`	Возвращает значение из перечисления `EntityState`, указывающее состояние объекта: `Added` (добавленный), `Deleted` (удаленный), `Detached` (отсоединенный), `Modified` (модифицированный) и `Unchanged` (неизменившийся)
`OriginalValues`	Возвращает коллекцию исходных значений свойств, индексированную по имени свойства
`CurrentValues`	Возвращает коллекцию текущих значений свойств, индексированную по имени свойства

Перечисленные свойства можно применять для инспектирования состояния сущностных объектов, что позволяет понять, каким образом работает процесс отслеживания изменений. Модифицируйте метод `UpdateProduct()` хранилища для вывода информации, касающейся отслеживания изменений, с использованием объекта `EntityEntry` (листинг 12.15).

Листинг 12.15. Инспектирование деталей, связанных с отслеживанием изменений, в файле `EFDataRepository.cs` из папки `Models`

```
using System;
using System.Collections.Generic;
using System.Linq;
using Newtonsoft.Json;
using Microsoft.EntityFrameworkCore.ChangeTracking;

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }

    // ...для краткости остальные методы не показаны...

    public void UpdateProduct(Product changedProduct) {
      Product originalProduct = context.Products.Find(changedProduct.Id);
      originalProduct.Name = changedProduct.Name;
      originalProduct.Category = changedProduct.Category;
      originalProduct.Price = changedProduct.Price;
      EntityEntry entry = context.Entry(originalProduct);
      Console.WriteLine($"Entity State: {entry.State}");
      foreach (string p_name in new string[]
          { "Name", "Category", "Price" }) {
        Console.WriteLine($"{p_name} - Old: " +
          $"{entry.OriginalValues[p_name]}, " +
          $"New: {entry.CurrentValues[p_name]}");
      }
      context.SaveChanges();
    }
```

```
    public void DeleteProduct(long id) {
      Console.WriteLine("DeleteProduct: " + id);
    }
  }
}
```

Модифицированный метод `UpdateProduct()` получает `EntityEntry` для объекта `Product`, созданного методом `Find()`, после чего выясняет значения свойств `State`, а также текущее и исходное значения для свойств `Name`, `Category` и `Price`.

Чтобы просмотреть информацию отслеживания, запустите приложение с помощью `dotnet run` и отредактируйте один из товаров, отображающихся в таблице. После щелчка на кнопке Save данные, отправленные браузером приложению, будут применены к сущностному объекту, и в окне командной строки появится информация отслеживания:

```
...
Entity State: Modified
Name - Old: Kayak, New: Green Kayak
Category - Old: Watersports, New: Watersports
Price - Old: 275.00, New: 275.00
...
```

Данные отслеживания показывают, что свойство `Name` товара `Kayak` было изменено на `Green Kayak`. Свойство `State` возвращает значение `Modified`, а инфраструктура Entity Framework Core запишет в БД только свойства, значения которых изменились, избегая обновления значений, оставшихся прежними.

Обновление в единственной операции базы данных

В предыдущем примере неизменившиеся свойства в БД не записывались, но это достигалось путем запрашивания у БД текущих значений, что просто обменивало одну неэффективность на другую. В настоящий момент вас может интересовать, что случилось с объектом `Product`, который использовался для заполнения полей HTML-формы, и почему его нельзя применять для обнаружения изменений?

Выбор стратегии обновления

Если ваша модель данных проста, а сущностные объекты имеют мало свойств, тогда вы должны использовать самую простую стратегию обновления и записывать полученные объекты в БД, как показано в листинге 12.13, даже когда это означает запись значений, которые не изменялись. Подход неэффективен, но неэффективность будет небольшой.

В случае более сложных моделей данных решение зависит от относительных стоимостей емкости сервера баз данных и сетевого трафика. Если основные затраты приходятся на сетевой трафик, то вы должны выполнять дополнительную операцию чтения БД, как демонстрировалось в листинге 12.14, что увеличит требования к БД, но сократит объем данных, отправляемых по сети в и из браузера. Если же основные затраты связаны с работой сервера баз данных, тогда вам следует включать исходные значения данных в HTML-форму, как делалось в листинге 12.17. Такой подход позволяет избежать операции БД, но за счет удвоения количества значений данных, включаемых в форму, и может потребовать средств для обнаружения ситуации, когда исходные данные изменяются другим пользователем, которые описаны в главе 21.

Каждый HTTP-запрос, принимаемый приложением, обрабатывается новым экземпляром контроллера Home, а каждый объект контроллера получает новый объект хранилища и новый объект контекста БД. После того как HTTP-запрос был обработан, объекты контроллера, хранилища и контекста БД отбрасываются наряду с любыми объектами Product, которые были извлечены из БД. Это означает, что каждый запрос обязан извлекать требующиеся ему данные из БД, даже когда более ранний HTTP-запрос от того же клиента применял те же самые запросы к БД.

Тем не менее, третья стратегия выполнения обновлений состоит в том, чтобы использовать в своих интересах первоначальную операцию чтения, включив полученные ею данные в ответ, который отправляется клиенту, и задействовать его для избегания обновлений неизмененных значений.

Первым делом отредактируйте файл Editor.cshtml, добавив скрытые элементы input, как показано в листинге 12.16.

Листинг 12.16. Добавление элементов в файле Editor.cshtml из папки Views/Shared

```
@model DataApp.Models.Product
@{
  ViewData["Title"] = ViewBag.CreateMode ? "Create" : "Edit";
  Layout = "_Layout";
}

<form asp-action="@(ViewBag.CreateMode ? "Create" : "Edit")"
      method="post">
  <input name="original.Id" value="@Model?.Id" type="hidden" />
  <input name="original.Name" value="@Model?.Name" type="hidden" />
  <input name="original.Category" value="@Model?.Category" type="hidden" />
  <input name="original.Price" value="@Model?.Price" type="hidden" />
  <div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
  </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>
```

Атрибуты name для скрытых элементов input снабжены префиксом original, за которым следует точка, что сообщает связывателю моделей MVC о том, что эти элементы должны применяться как свойства для параметра метода действия, чьим именем является original. Такие элементы предоставят исходные значения данных, когда браузер отправит HTTP-запрос POST.

Чтобы получить исходные данные, добавьте к методу `Edit()` контроллера `Home` параметр (листинг 12.17). Параметр должен иметь имя `original`, которое соответствует элементам `input` в листинге 12.16, и связыватель моделей MVC создаст объект `Product` с использованием значений из этих элементов `input`, снабжая приложение легким доступом к исходным значениям.

Листинг 12.17. Привязка исходных значений в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index(string category = null, decimal? price = null)
    {
      var products = repository.GetFilteredProducts(category, price);
      ViewBag.category = category;
      ViewBag.price = price;
      return View(products);
    }

    public IActionResult Create() {
      ViewBag.CreateMode = true;
      return View("Editor", new Product());
    }

    [HttpPost]
    public IActionResult Create(Product product) {
      repository.CreateProduct(product);
      return RedirectToAction(nameof(Index));
    }

    public IActionResult Edit(long id) {
      ViewBag.CreateMode = false;
      return View("Editor", repository.GetProduct(id));
    }

    [HttpPost]
    public IActionResult Edit(Product product, Product original) {
      repository.UpdateProduct(product, original);
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Delete(long id) {
      repository.DeleteProduct(id);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Связыватель моделей MVC создаст два объекта Product: один объект будет содержать значения из элементов input, которые могли быть отредактированы пользователем, а другой объект будет содержать исходные значения. Модифицированный метод Edit() передает оба объекта хранилищу, которое потребуется обновить, чтобы оно получало их. Модифицируйте метод UpdateProduct(), определенный интерфейсом IDataRepository, как демонстрируется в листинге 12.18.

Листинг 12.18. Обновление метода в файле IDataRepository.cs из папки Models

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

namespace DataApp.Models {
  public interface IDataRepository {

    Product GetProduct(long id);

    IEnumerable<Product> GetAllProducts();

    IEnumerable<Product> GetFilteredProducts(string category = null,
      decimal? price = null);

    void CreateProduct(Product newProduct);

    void UpdateProduct(Product changedProduct, Product originalProduct = null);

    void DeleteProduct(long id);
  }
}
```

Чтобы отслеживать изменения, добавьте к методу UpdateProduct() класса реализации EFDataRepository необязательный параметр и примените его для обнаружений изменений (листинг 12.19).

Листинг 12.19. Отслеживание изменений в файле EFDataRepository.cs из папки Models

```
...
public void UpdateProduct(Product changedProduct,
                          Product originalProduct = null) {
  if (originalProduct == null) {
    originalProduct = context.Products.Find(changedProduct.Id);
  } else {
    context.Products.Attach(originalProduct);
  }
  originalProduct.Name = changedProduct.Name;
  originalProduct.Category = changedProduct.Category;
  originalProduct.Price = changedProduct.Price;
  context.SaveChanges();
}
...
```

Если метод UpdateProduct() получает параметр originalProduct, тогда он помещается под управление инфраструктуры Entity Framework Core, используя метод DbSet<T>.Attach(), который настраивает процесс отслеживания изменений Entity Framework Core и устанавливает ассоциированное свойство EntityEntry.State в Unmodified.

Значения свойств из другого объекта Product копируются в отслеживаемый объект. Процесс обнаружения изменений гарантирует, что в БД будут сохраняться только измененные значения.

Перезапустите приложение, снова щелкните на кнопке Edit для товара Stadium и измените значение Stadium (Small) в поле Name на Stadium (Regular). После щелчка на кнопке Save в журнальных сообщениях появится SQL-оператор, обновляющий только свойство, которое было изменено:

```
...
UPDATE [Products] SET [Name] = @p0
WHERE [Id] = @p1;
...
```

Для выяснения, какое свойство изменилось, дополнительный запрос к БД не требуется, поскольку исходные данные были включены в HTTP-запрос POST.

Совет. Подробные сведения о том, как обнаруживать ситуации, когда исходные данные изменились с момента их чтения из БД, приведены в главе 21.

Удаление данных

Последней операцией над данными, которую предстоит реализовать, будет удаление; по сравнению с обновлением это относительно простой процесс. Чтобы добавить поддержку удаления строк из БД, отредактируйте метод DeleteProduct() в классе хранилища согласно листингу 12.20.

Листинг 12.20. Удаление данных в файле EFDataRepository.cs из папки Models

```
...
public void DeleteProduct(long id) {
  Product p = context.Products.Find(id);
  context.Products.Remove(p);
  context.SaveChanges();
}
...
```

В классе DbSet определен метод Remove(), который принимает сущностный объект. Когда вызывается метод SaveChanges(), инфраструктура Entity Framework Core сообщает серверу баз данных о необходимости удалить строку из таблицы БД. Просмотрев вывод из приложения, вы увидите SQL-оператор, в который транслируется вызов метода Remove():

```
...
DELETE FROM [Products]
  WHERE [Id] = @p0;
...
```

Проблема в листинге 12.20 связана с применением метода Find() для запрашивания БД с целью получения объекта Product, подлежащего удалению. Прием работает, но в результате дает операцию БД, которой можно избежать, создавая объект Product напрямую (листинг 12.21).

Совет. Свойство `DbSet<T>` также определяет метод `RemoveRange()`, который можно использовать для удаления множества объектов в одном вызове, как объясняется в главе 16.

Листинг 12.21. Удаление данных в файле `EFDataRepository.cs` из папки `Models`

```
...
public void DeleteProduct(long id) {
  context.Products.Remove(new Product { Id = id });
  context.SaveChanges();
}
...
```

Для идентификации удаляемой из БД строки применяется только ключ, поэтому операция удаления может быть выполнена путем создания нового объекта `Product` с одним лишь значением `Id` и его передачи методу `Remove()`. Чтобы увидеть эффект, перезапустите приложение, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete (Удалить) для удаления товара из БД (рис. 12.6). Результат будет тем же, но без необходимости в чтении данных перед их удалением.

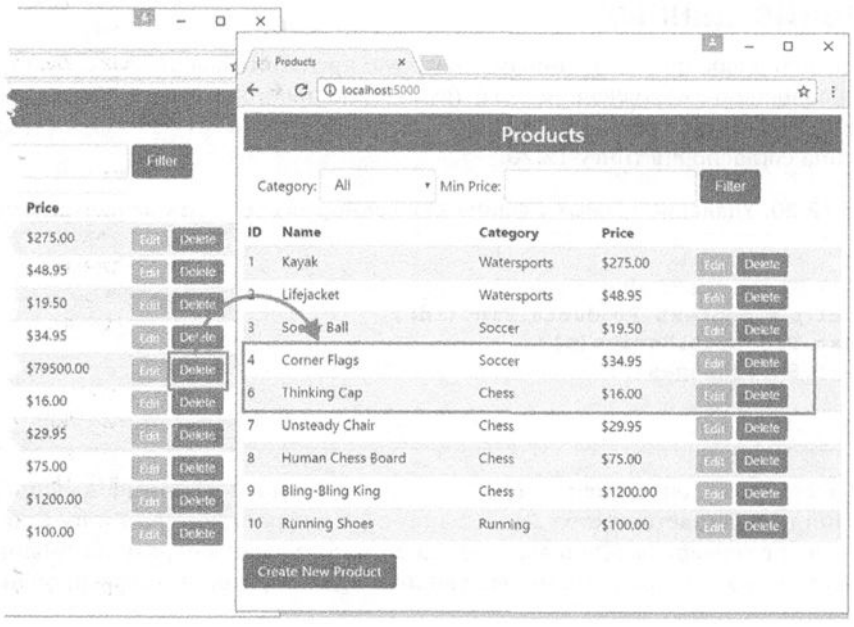

Рис. 12.6. Удаление данных

Резюме

В главе приводились объяснения, каким образом выполнять основные операции над данными, используя Entity Framework Core. Большей частью такие операции просты в выполнении и хорошо вписываются в модель MVC, хотя потребуется уделить внимание запросам, которые применяются для выполнения каждой задачи — частая тема при использовании инфраструктуры Entity Framework Core. В следующей главе будет описано средство миграций Entity Framework Core, применяемое для подготовки БД к хранению данных приложения.

ГЛАВА 13

Работа с миграциями

В этой главе рассматривается средство миграций — способ, которым инфраструктура Entity Framework Core гарантирует, что БД отражает модель данных в приложении, даже когда модель изменяется. Подход известен под названием *"сначала код"*, при котором вы начинаете с требуемых классов модели данных и используете миграции для создания и управления БД. (Противоположный подход, называемый *"сначала БД"*, предусматривает работу с существующей БД и описан в главах 17 и 18.)

В главе объясняется, как создавать и применять миграции, каким образом оценивать влияние миграции и как управлять миграциями с использованием инструментов командной строки и API-интерфейса. В табл. 13.1 приведены сведения, позволяющие поместить миграции в контекст.

Таблица 13.1. Помещение миграций в контекст

Вопрос	Ответ
Что это такое?	Миграции представляют собой группы команд, которые выполняют подготовку БД для применения в приложениях Entity Framework Core. Они используются для создания БД и затем поддерживаются в синхронизированном состоянии с изменениями, вносимыми в модель данных
Чем они полезны?	Миграции автоматизируют процесс создания и сопровождения БД для хранения данных приложений. В отсутствие миграций пришлось бы создавать БД с помощью SQL-команд и вручную конфигурировать инфраструктуру Entity Framework Core для работы с БД
Как они используются?	Миграции создаются и применяются с использованием инструментов командной строки `dotnet ef`
Существуют ли какие-то скрытые ловушки или ограничения?	Сообщения об ошибках, которые выдает Entity Framework Core при работе с миграциями, не всегда ясны. Самая распространенная проблема связана с тем, что миграции создаются, но не применяются к БД
Существуют ли альтернативы?	Можно сначала создать БД, а затем классы модели данных для использования с инфраструктурой Entity Framework Core (примеры приведены в главе 17)

Внимание! Многие команды, описанные в главе, изменяют структуру БД и могут привести к потере данных. Не применяйте эти команды в производственных БД, пока четко не осознаете их последствия и не создадите резервные копии критически важных данных.

В табл. 13.2 приведена сводка по главе.

Таблица 13.2. Сводка по главе

Задача	Решение	Листинг
Создание новой миграции	Используйте команду `dotnet ef migration add`	13.12
Просмотр изменений, которые содержит миграция	Применяйте команду `dotnet ef migrations script`	13.3, 13.10, 13.13
Применение миграции к БД	Используйте команду `dotnet ef database update`	13.4–13.9, 15–17
Получение списка миграций в проекте	Применяйте команду `dotnet ef migrations list`	13.14
Удаление миграции	Используйте команду `dotnet ef migrations remove`	13.18–13.20
Переустановка БД	Применяйте команду `dotnet ef database update 0` или `dotnet ef database drop`	13.21, 13.22
Управление миграциями для множества БД	Создайте отдельный класс контекста и укажите его имя при использовании команды `dotnet ef`	13.23–13.28, 13.30, 13.31
Просмотр контекстов БД в проекте	Используйте команду `dotnet ef dbcontext list`	13.29
Программное управление миграциями	Применяйте API-интерфейс миграций Entity Framework Core	13.32–13.35
Программное заполнение БД начальными данными	Используйте API-интерфейс миграций, чтобы гарантировать отсутствие ожидающих обновлений перед использованием класса контекста для добавления данных в БД	13.36–13.39

Подготовительные шаги

В настоящей главе используется проект `DataApp`, созданный в главе 11 и модифицированный в главе 12. Чтобы провести подготовку, откройте окно командной строки или PowerShell, перейдите в папку проекта `DataApp` и введите команду из листинга 13.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 13.1. Удаление БД

```
dotnet ef database drop --force
```

Команда удаляет БД, что поможет вам получить ожидаемые результаты в последующих примерах. Запустите приложение, выполнив команду из листинга 13.2.

Листинг 13.2. Запуск примера приложения

```
dotnet run
```

Приложение запустится, но при переходе в браузере по ссылке `http://localhost:5000` возникнет ошибка. Трассировка стека оказывается длинной, и ошибка повторяется несколько раз, а вот как выглядит важная часть сообщения:

```
...
SqlException: Cannot open database "DataAppDb" requested by the login.
SqlException: Не удалось открыть базу данных DataAppDb, запрошенную входом.
...
```

Понятие миграций

Причина ошибки, описанной в предыдущем разделе, связана с тем, что БД не была подготовлена для приложения. В строке подключения внутри файла `appsettings.json` указывается БД по имени `DataAppDb`:

```
...
"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=DataAppDb;
  MultipleActiveResultSets=true"
...
```

В настоящий момент серверу SQL Server ничего не известно о БД по имени `DataAppDb`, так что при попытке приложения прочитать из нее данные возникает ошибка. Подготовка БД производится с помощью *миграций* Entity Framework Core. Миграция — это класс C#, который содержит инструкции для создания схемы БД. Когда миграция применяется, создаются БД, таблицы и столбцы, требующиеся для хранения сущностных данных.

Работа с начальной миграцией

В примере приложения уже имеется миграция, которая была создана в главе 11 и использовалась приложением в предшествующих главах. Откройте папку `DataApp/Migrations` и вы заметите три файла. (Один из файлов является вложенным элементом в окне Solution Explorer и не будет виден до тех пор, пока не раскроется его родительский элемент.) Все три файла описаны в табл. 13.3.

Таблица 13.3. Файлы миграции в проекте примера приложения

Имя	Описание
`<отметка времени>_Initial.cs`	Часть класса `Initial`, которая применяет первую миграцию к БД. Содержит инструкции для создания схемы БД
`<отметка времени>_Initial.Designer.cs`	Часть класса `Initial`, которая применяет первую миграцию к БД. Содержит инструкции для создания объектов модели
`EFDatabaseContextModelSnapshot.cs`	Этот класс содержит описание сущностных классов, используемых в миграции, и применяется для обнаружения изменений при создании будущих миграций

Первые два файла представляют собой частичные классы, т.е. содержащийся в них код объединяется для формирования единого класса C#. Имена классов начинаются с отметки времени, указывающей, когда они были созданы, за которой следует имя содержащейся в них миграции. В данном случае имена включают `_Initial`, поскольку миграция создавалась в главе 11 с использованием такой команды (выполнять ее еще раз не требуется):

```
dotnet ef migrations add Initial
```

По соглашению для первой миграции, созданной в проекте, выбирается имя `Initial`. Откройте файл `<отметка времени>_Initial.cs` и вы увидите самую важную часть миграции:

```
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;
using System;
using System.Collections.Generic;

namespace DataApp.Migrations {
  public partial class Initial : Migration {

    protected override void Up(MigrationBuilder migrationBuilder) {

      migrationBuilder.CreateTable(
        name: "Products",
        columns: table => new {
          Id = table.Column<long>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
              SqlServerValueGenerationStrategy.IdentityColumn),
          Category = table.Column<string>(nullable: true),
          Name = table.Column<string>(nullable: true),
          Price = table.Column<decimal>(nullable: false)
        },
        constraints: table => {
          table.PrimaryKey("PK_Products", x => x.Id);
        });
    }

    protected override void Down(MigrationBuilder migrationBuilder) {
      migrationBuilder.DropTable(
        name: "Products");
    }
  }
}
```

На заметку! Точный код может меняться, особенно если вы используете более новую версию инфраструктуры Entity Framework Core или поставщика БД для SQL Server. Однако назначение и общая природа кода будут достаточно близки, чтобы понять, что делает миграция.

Показанная часть класса `Initial` содержит методы, которые будут вызываться для обновления БД. Каждая миграция содержит два метода, `Up()` и `Down()`, которые описаны в табл. 13.4.

Таблица 13.4. Методы миграции

Имя	Описание
`Up()`	Содержит операторы, которые повышают БД для хранения сущностных данных
`Down()`	Содержит операторы, которые понижают БД до исходного состояния

Миграцию можно применять для повышения или понижения БД — процесса, который вскоре станет более понятным. В случае недавно созданной БД процесс повышения создаст таблицу со столбцами, необходимыми для хранения данных, тогда как процесс понижения возвратит БД в исходное состояние. Оба процесса будут описаны в последующих разделах.

Другие команды `dotnet ef`

Работа с Entity Framework Core означает использование командной строки для создания и применения миграций. Все команды, задействованные в книге, начинаются с `dotnet ef`, подобно следующей команде из главы 11:

```
dotnet ef migration add Initial
```

Команда создает новую миграцию, которая применяется к БД с использованием похожей команды, показанной в листинге 13.4 далее в главе:

```
dotnet ef database update
```

Если вы имели дело с более ранними версиями Entity Framework, то можете быть знакомы с другим набором команд, таким как `Add-Migration` и `Update-Database`. Это были первоначальные команды, применяемые для управления миграциями и БД, они поддерживались только средой Visual Studio и надежно работали лишь при использовании в консоли диспетчера пакетов (Package Manager Console), представляющей собой окно PowerShell с рядом дополнительных возможностей.

Такой стиль команд по-прежнему поддерживается, но в книге они не применяются, поскольку работают только в специфическом окне Visual Studio и вызывают бесчисленные проблемы.

Процесс повышения

Метод `Up()` отвечает за повышение БД. При обработке миграции, которая применяется к недавно созданной БД, инфраструктура Entity Framework Core создает таблицу для каждого свойства `DbSet<T>` в классе контекста. В настоящий момент в контексте имеется только одно свойство `DbSet` по имени `Products`, которое дает в результате следующий оператор в миграции:

```
...
migrationBuilder.CreateTable(
  name: "Products",
...
```

Параметром метода `Up()` является экземпляр класса `MigrationBuilder`, предоставляющий методы, которые используются для указания изменений, подлежащих применению к БД. Поставщик БД преобразует эти методы в команды, специфичные для БД, что и представляет собой способ трансляции операторов C# миграции в операторы SQL, которые способен выполнять сервер SQL Server.

Метод `MigrationBuilder.CreateTable()` создает новую таблицу БД. По умолчанию для имени таблицы БД инфраструктура Entity Framework Core использует имя свойства `DbSet<T>`, чем и объясняется установка аргумента `name` метода `CreateTable()` в `Products` внутри миграции.

Остальные аргументы метода `CreateTable()` конфигурируют таблицу. Аргумент `columns` определяет столбец в таблице для каждого свойства, определенного сущностным классом. Свойство `Products`, определенное классом контекста, имеет тип `DbSet<Product>`, поэтому Entity Framework Core добавляет в таблицу столбцы для всех свойств, определенных классом `Product`:

```
...
migrationBuilder.CreateTable(
  name: "Products",
  columns: table => new {
    Id = table.Column<long>(nullable: false)
      .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn),
    Category = table.Column<string>(nullable: true),
    Name = table.Column<string>(nullable: true),
    Price = table.Column<decimal>(nullable: false)
  },
...
```

Здесь определяются столбцы для свойств `Id`, `Name`, `Category` и `Price`. Свойство `Id` конфигурируется так, что его значения будут генерироваться БД при добавлении новых строк в таблицу. Каждая строка конфигурируется с типом .NET для соответствующего свойства `Product`.

В конце определения таблицы добавляется ограничение, которое назначает столбец `Id` первичным ключом:

```
...
migrationBuilder.CreateTable(
  name: "Products",
  columns: table => new {
    Id = table.Column<long>(nullable: false)
      .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn),
    Category = table.Column<string>(nullable: true),
    Name = table.Column<string>(nullable: true),
    Price = table.Column<decimal>(nullable: false)
  },
  constraints: table => {
    table.PrimaryKey("PK_Products", x => x.Id);
  });
}
...
```

Собрав все фрагменты вместе, метод `Up()` создает новую таблицу по имени `Products` со столбцами `Id`, `Name`, `Category` и `Price`, сконфигурированную так, что столбец `Id` становится первичным ключом, а значения для него будут генерироваться БД при добавлении новых строк.

Процесс понижения

Метод `Down()` предназначен для возвращения БД в предыдущее состояние, аннулируя эффект от метода `Up()`. Когда речь идет о миграции, которая будет применена к пустой БД, понижение просто удаляет таблицу, созданную для хранения сущностных данных:

```
...
migrationBuilder.DropTable(name: "Products");
...
```

Метод `MigrationBuilder.DropTable()` удаляет таблицу из БД. В рассматриваемом случае будет удалена таблица `Products`, что возвратит БД в ее исходное состояние.

Исследование SQL-операторов миграции

Чтобы посмотреть, как операторы C# класса миграции `Initial` транслируются в операторы SQL, выполните в окне командной строки команду из листинга 13.3, находясь в папке `DataApp`.

Листинг 13.3. Исследование миграции

```
dotnet ef migrations script
```

Все инструменты командной строки Entity Framework Core вызываются с помощью команды `dotnet ef`, которая нацеливается на инструменты EF Core, добавленные в главе 11. Аргумент `migrations` выбирает инструменты, выполняющие операции на миграциях, а аргумент `script` позволяет отобразить SQL-команды, которые миграция будет выполнять в БД. Для миграции `Initial` в примере приложения сгенерируется следующий SQL-сценарий:

```
...
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
  CREATE TABLE [__EFMigrationsHistory] (
    [MigrationId] nvarchar(150) NOT NULL,
    [ProductVersion] nvarchar(32) NOT NULL,
    CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
  );
END;

GO

CREATE TABLE [Products] (
  [Id] bigint NOT NULL IDENTITY,
  [Category] nvarchar(max) NULL,
  [Name] nvarchar(max) NULL,
  [Price] decimal(18, 2) NOT NULL,
  CONSTRAINT [PK_Products] PRIMARY KEY ([Id])
);

GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20180124114307_Initial', N'2.0.1-rtm-125');

GO
...
```

Приведенные операторы создают две таблицы. Таблица по имени Products соответствует инструкциям, содержащимся в миграции, которая была описана в предыдущем разделе. Таблица по имени __EFMigrationsHistory используется для отслеживания, какие миграции применялись к БД; ее важность прояснится при создании дополнительных миграций позже в главе.

Применение миграции

Миграция не дает никакого эффекта до тех пор, пока она не будет применена к БД. Выполните команду из листинга 13.4 в папке DataApp, чтобы применить миграцию к БД.

Листинг 13.4. Применение миграции к БД

```
dotnet ef database update
```

Команда dotnet ef нацеливается на инструменты Entity Framework Core; аргумент database указывает, что нужно выполнить операцию в БД, а аргумент update сообщает Entity Framework Core о необходимости обновления БД за счет применения всех миграций в проекте (хотя пока есть лишь одна миграция). Проект скомпилируется и код в методе Up() класса Initial выполнится для генерации SQL-команд, которые создают и конфигурируют таблицу Products.

Совет. В случае применения миграции с помощью команды dotnet ef database update указанная в строке подключения БД будет создана, если она не существует. Именно так создается БД по имени DataAppDb, на использование которой сконфигурировано приложение, даже когда имя БД не является частью миграции создаваемого сценария SQL.

Проверка миграции

Выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) и введите (localdb)\MSSQLLocalDB в поле Server Name (Имя сервера). Удостоверьтесь в том, что в поле Authentication (Аутентификация) выбран вариант Windows Authentication (Аутентификация Windows) и щелкните на меню Database Name (Имя базы данных), чтобы выбрать DataAppDb в списке, который отобразит имена баз данных, созданных с использованием LocalDB. После подключения к БД введите в окне редактора код SQL, приведенный в листинге 13.5.

Листинг 13.5. Инспектирование таблицы Products

```
USE DataAppDb

SELECT column_name, data_type FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'Products'

GO
```

Выберите пункт Execute (Выполнить) в меню SQL среды Visual Studio, чтобы отправить запрос серверу баз данных. Результат содержит имя и тип каждого столбца в таблице Products, отражая результат миграции (табл. 13.5), и показывает, что строка в таблице Products содержит столбцы для всех значений данных, необходимых для представления объекта Product.

Таблица 13.5. Структура таблицы Products

column_name	data_type
Id	bigint
Category	nvarchar
Name	nvarchar
Price	decimal

Заполнение базы данных начальными данными и выполнение приложения

Теперь, когда схема БД создана, инфраструктура Entity Framework Core будет считывать и сохранять объекты Product по поручению приложения. Чтобы снабдить приложение данными для работы, введите в текущем окне редактора (или откройте новое) код SQL из листинга 13.6.

Листинг 13.6. Заполнение БД начальными данными

```
USE DataAppDb

INSERT INTO Products (Name, Category, Price)
VALUES
  ('Kayak', 'Watersports', 275),
  ('Lifejacket', 'Watersports', 48.95),
  ('Soccer Ball', 'Soccer', 19.50),
  ('Corner Flags', 'Soccer', 34.95),
  ('Stadium', 'Soccer', 79500),
  ('Thinking Cap', 'Chess', 16),
  ('Unsteady Chair', 'Chess', 29.95),
  ('Human Chess Board', 'Chess', 75),
  ('Bling-Bling King', 'Chess', 1200)
```

Выберите пункт Execute в меню SQL; вы получите следующий ответ, который сообщает количество строк, измененных командой:

```
(9 row(s) affected)
(9 строк(а) затронуто)
```

Заполнение БД начальными данными вручную — несколько неудобный процесс, но позже в главе будет показано, как обработать его с применением кода C#. Выполните команду из листинга 13.7 в папке DataApp, чтобы запустить пример приложения.

Листинг 13.7. Запуск примера приложения

```
dotnet run
```

Откройте новое окно браузера и перейдите по ссылке http://localhost:5000. Вы увидите таблицу данных (рис. 13.1). Убедившись в том, что БД работоспособна и заполнена начальными данными, остановите приложение с помощью <Ctrl+C>.

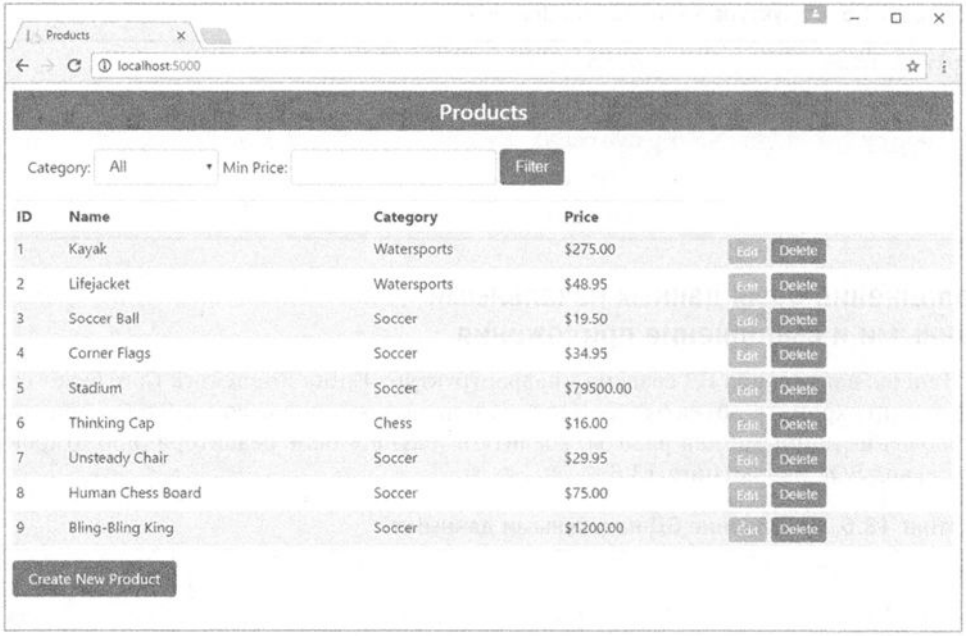

Рис. 13.1. Выполнение примера приложения

Прежде чем двигаться дальше, стоит поразмышлять об эффекте миграции БД. В начале этого раздела БД отсутствовала, а потому не было способа хранения объектов `Product`. Созданная в главе 11 миграция представляет собой набор файлов C#, которые содержат инструкции для повышения и понижения БД. В результате применения миграции была создана БД по имени `DataAppDb`, и операторы C# для повышения БД превратились в SQL-команды, выполнение которых привело к созданию и настройке таблицы `Products` со столбцами для всех свойств, определенных классом `Product`. Каждая строка в таблице `Products` олицетворяет объект `Product`, начиная с первоначальных данных в листинге 13.6.

Полезно также подумать о том, до какой степени процесс был автоматическим. Инфраструктура Entity Framework Core обнаружила класс контекста БД, идентифицировала классы, подлежащие хранению в БД, и выяснила, как представлять экземпляры этих классов в виде реляционных данных. Как вы увидите в последующих главах, для более сложных примеров инфраструктура Entity Framework Core может требовать соблюдения некоторого направления, но вы вольны избрать длинный путь, просто определив классы C# и позволив Entity Framework Core позаботиться о деталях.

Создание дополнительных миграций

Удобство миграций становится очевидным, когда модель данных в приложении MVC изменяется. Чтобы взглянуть, каким образом Entity Framework Core обрабатывает изменения в модели данных, добавьте в класс `Product` перечисление и свойство (листинг 13.8).

Листинг 13.8. Добавление свойства в файле `Product.cs` из папки `Models`

```
namespace DataApp.Models {
  public enum Colors {
    Red, Green, Blue
  }
  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    public Colors Color { get; set; }
  }
}
```

Свойство `Color` использует перечисление `Colors`. Добавление свойства означает, что объекты `Product` больше не могут сохраняться в БД, поскольку отсутствуют средства для хранения значений `Color`. Приведение БД в синхронизированное состояние с видоизмененной моделью данных означает создание и применение новой миграции. Для создания миграции выполните в папке `DataApp` команду из листинга 13.9.

Листинг 13.9. Создание новой миграции БД

```
dotnet ef migrations add AddColorProperty
```

Команда `dotnet ef migrations add` создает новую миграцию по имени `AddColorProperty`. Вы можете выбирать для миграций любые желаемые имена, но распространенная стратегия именования предполагает включение описания изменения, внесенного в модель данных, либо использование увеличивающихся номеров версий.

В результате выполнения команды из листинга 13.9 в папку `Migrations` добавляются новые файлы. Откройте файл `<отметка времени>_AddColorProperty.cs`, чтобы просмотреть, какие изменения миграция применит к БД:

```
using Microsoft.EntityFrameworkCore.Migrations;
using System;
using System.Collections.Generic;
namespace DataApp.Migrations {
  public partial class AddColorProperty : Migration {
    protected override void Up(MigrationBuilder migrationBuilder) {
      migrationBuilder.AddColumn<int>(
        name: "Color",
        table: "Products",
        nullable: false,
        defaultValue: 0);
    }
    protected override void Down(MigrationBuilder migrationBuilder) {
      migrationBuilder.DropColumn(
        name: "Color",
       table: "Products");
  }
 }
}
```

Метод `Up()` обновит схему, отразив изменение в модели данных, что в рассматриваемом случае означает добавление в таблицу `Products` столбца по имени `Color`. Метод `Down()` возвращает БД в предыдущее состояние, удаляя столбец `Color`.

Выполните в папке `DataApp` команду из листинга 13.10 для просмотра SQL-операторов, которые выпустит новая миграция.

Листинг 13.10. Просмотр SQL-операторов миграции

```
dotnet ef migrations script Initial AddColorProperty
```

Указание имен миграций в команде `dotnet ef migrations script` приводит к выводу операторов, требующихся для обновления БД от первой миграции до второй:

```
...
ALTER TABLE [Products] ADD [Color] int NOT NULL DEFAULT 0;

GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'<отметка времени>_AddColorProperty', N'2.0.1-rtm-125');

GO
...
```

Показанные выше операторы добавляют столбец `Color` в таблицу `Products` и обновляют таблицу `__EFMigrationsHistory`, отражая факт применения миграции к БД.

Добавление в модель данных еще одного свойства

Для демонстрации управления изменениями в БД понадобится создать третью миграцию. Добавьте в класс `Product` свойство, показанное в листинге 13.11.

Листинг 13.11. Добавление свойства в файле `Product.cs` из папки `Models`

```
namespace DataApp.Models {

  public enum Colors {
    Red, Green, Blue
  }

  public class Product {

    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    public Colors Color { get; set; }
    public bool InStock { get; set; }
  }
}
```

Новое свойство имеет тип `bool` и называется `InStock`. Чтобы создать миграцию, которая добавит поддержку хранения этого свойства в БД, выполните команду из листинга 13.12, находясь в папке `DataApp`.

Листинг 13.12. Создание еще одной миграции

```
dotnet ef migrations add AddInStockProperty
```

В папку Migrations проекта добавится новый набор файлов миграции. Выполните в папке DataApp команду из листинга 13.13, чтобы просмотреть SQL-операторы, которые будут модифицировать таблицу Products.

Листинг 13.13. Отображение SQL-операторов для миграции

```
dotnet ef migrations script AddColorProperty AddInStockProperty
```

Команда дает следующий результат:

```
...
ALTER TABLE [Products] ADD [InStock] bit NOT NULL DEFAULT 0;

GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'<отметка времени>_AddInStockProperty', N'2.0.1-rtm-125');

GO
...
```

Легко заметить, что каждое изменение, внесенное в класс Product, производит миграцию, которая приводит модель данных и БД снова в синхронизированное состояние. При создании миграции инфраструктура Entity Framework Core автоматически определяет, какие изменения должны быть применены к БД, чтобы миграция вносила только модификации, которые необходимы для представления экземпляров видоизмененных классов.

Управление миграциями

Создание миграций — только часть процесса: миграция должна быть применена к БД, чтобы обеспечить возможность сохранения данных приложения. Наиболее распространенный способ управления миграциями предусматривает использование инструментов командной строки dotnet ef, которые рассматриваются в последующих разделах. Миграциями можно также управлять с помощью API-интерфейса, предлагаемого инфраструктурой Entity Framework Core, который будет описан в разделе "Программное управление миграциями" далее в главе.

Вывод списка миграций

Просмотреть список миграций, созданных в проекте, можно путем выполнения в папке DataApp команды из листинга 13.14.

Листинг 13.14. Вывод списка доступных миграций

```
dotnet ef migrations list
```

Команда выводит список всех миграций, которые были созданы для проекта. В проекте DataApp доступны три миграции, что дает в результате такой вывод (отметки времени в именах ваших файлов могут быть другими):

```
...
<отметка времени>_Initial
<отметка времени>_AddColorProperty
<отметка времени>_AddInStockProperty
...
```

Список содержит миграции, определенные в проекте, которые возможно еще не были применены к БД. Миграции перечислены по порядку, т.е. можно легко понять, что миграция AddColorProperty основана на изменениях, содержащихся в миграции Initial, а миграция AddInStockProperty построена на основе миграции AddColorProperty.

Применение всех миграций

Самой распространенной задачей является применение всех миграций, определенных в проекте, для приведения БД в актуальное состояние за один шаг. Чтобы применить все миграции в проекте DataApp, выполните команду, показанную в листинге 13.15, внутри папки DataApp.

Листинг 13.15. Применение всех миграций, определенных в проекте

```
dotnet ef database update
```

С помощью таблицы __EFMigrationsHistory инфраструктура Entity Framework Core выяснит, какие миграции уже применялись к БД, и приведет БД в актуальное состояние.

Выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) и введите SQL-код из листинга 13.16.

Листинг 13.16. Просмотр таблицы Products

```
USE DataAppDb

SELECT column_name, data_type FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'Products'

GO
```

Выберите пункт Execute (Выполнить) в меню SQL среды Visual Studio, чтобы отправить запрос серверу баз данных. Результатом будет сводка по структуре таблицы (табл. 13.6), включающая столбцы, которые соответствуют новым свойствам, добавленным в класс Product в листингах 13.8 и 13.11.

Таблица 13.6. Структура таблицы Products

column_name	data_type
Id	bigint
Category	nvarchar
Name	nvarchar
Price	decimal
Color	int
InStock	bit

Обновление до специфической миграции

Можно провести обновление БД до специфической миграции, что полезно, когда необходимо откатить набор изменений или нежелательно применять все миграции за один раз. Выполните в папке DataApp команду из листинга 13.17, чтобы обновить БД до миграции AddColorProperty.

Листинг 13.17. Обновление БД до специфической миграции

```
dotnet ef database update AddColorProperty
```

Когда в команде dotnet ef database update указано имя, инфраструктура Entity Framework Core исследует БД с целью выяснения, какие миграции были применены, и начинает работать в направлении целевой миграции, вызывая метод Up() или Down() для выполнения повышений или понижений, требуемых для достижения целевого состояния.

В случае команды из листинга 13.17 для обновления до миграции AddColorProperty инфраструктуре Entity Framework Core придется выполнить понижение от миграции AddInStockProperty, что приведет к удалению столбца InStock из таблицы Products. Снова выполнив SQL-запрос из листинга 13.16, можно увидеть изменение в структуре таблицы (табл. 13.7).

Таблица 13.7. Структура таблицы Products после обновления до миграции AddColorProperty

column_name	data_type
Id	bigint
Category	nvarchar
Name	nvarchar
Price	decimal
Color	int

Удаление миграции

Временами миграции создаются по ошибке или перестают быть полезными. Выполните в папке DataApp команду, приведенную в листинге 13.18, чтобы удалить самую последнюю миграцию, добавленную в проект.

Листинг 13.18. Удаление самой последней миграции

```
dotnet ef migrations remove
```

Команда из листинга 13.18 удаляет миграцию AddInStockProperty, как видно по ее выводу:

```
...
Removing migration '<timestamp>_AddInStockProperty'.
Reverting model snapshot.
Удаление миграции <отметка времени>_AddInStockProperty.
Возвращение к прежнему состоянию снимка модели.
...
```

Сообщение о возвращении к прежнему состоянию снимка модели относится к файлу `EFDatabaseContextModelSnapshot.cs` внутри папки `Migrations`, который используется для сравнения сущностных классов в проекте с самыми последними миграциями. В случае удаления миграции снимок обновляется, чтобы отразить изменение. Это особенно полезно в ситуации, когда миграция удаляется из-за того, что ее имя было записано некорректно, поскольку миграция, создаваемая следующей, будет отражать любые изменения в классах модели с момента последней оставшейся миграции в проекте.

Совет. Удалять можно только самую последнюю миграцию, а это значит, что если необходимо удалить множество миграций, то команду из листинга 13.18 понадобится выполнить несколько раз.

Если миграция, которую вы собираетесь удалить, уже была применена к БД, тогда появится предупреждение. Оно очень важно, т.к. возникает риск перевести БД в состояние, откуда ее не удастся повысить из-за того, что схема не совпадает с состоянием, которое ожидают обнаружить оставшиеся миграции. Чтобы увидеть предупреждение, выполните в папке `DataApp` команду, представленную в листинге 13.19.

Листинг 13.19. Попытка удаления примененной миграции

```
dotnet ef migrations remove
```

Команда сообщает инфраструктуре Entity Framework Core о том, что нужно удалить миграцию `AddColorProperty`, которая была применена к БД. Отобразится следующее предупреждение, а миграция не удалится:

```
...
The migration '<timestamp>_AddColorProperty' has already been applied
to the database.
Revert it and try again. If the migration has been applied to other
databases, consider reverting its changes using a new migration.
```

Миграция <отметка времени>_AddColorProperty уже была применена к БД. Верните ее в прежнее состояние и повторите попытку. Если миграция была применена к другим БД, обдумайте отмену ее изменений с использованием новой миграции.

```
...
```

Вы можете воспользоваться командой `dotnet ef database update` и удалить миграцию из БД либо указать аргумент `--force` (листинг 13.20), чтобы сообщить инфраструктуре Entity Framework Core о необходимости продолжить в любом случае и удалить миграцию из проекта. Находясь в папке `DataApp`, выполните команду из листинга 13.20 для принудительного удаления миграции, несмотря на то, что она была применена к БД.

Листинг 13.20. Принудительное удаление миграции, примененной к БД

```
dotnet ef migrations remove --force
```

Когда вы инициируете принудительное удаление миграции, инфраструктура Entity Framework Core не проверяет состояние БД перед удалением миграции из проекта, что подтверждается выводом команды:

```
...
Removing migration '<timestamp>_AddColorProperty' without checking
the database.
If this migration has been applied to the database, you will need to
manually reverse the changes it made.
Removing migration '<timestamp>_AddColorProperty'.
Reverting model snapshot.
Удаление миграции <отметка времени>_AddColorProperty' без проверки БД.
Если эта миграция применялась к БД, тогда вам придется вручную
отменить внесенные ею изменения.
Удаление миграции <отметка времени>_AddColorProperty.
Возвращение к прежнему состоянию снимка модели.
...
```

Переустановка базы данных

Возникают ситуации, когда нужно отменить все ранее примененные к БД миграции и начать сначала. Причиной может быть проверка того, что миграции производят ожидаемую схему, или же принудительное удаление миграции, которая была применена к БД, из-за чего схема и проект перестали находиться в синхронном состоянии.

Введите в папке DataApp команду, показанную в листинге 13.21, чтобы выполнить инструкции в методах Down() всех миграций в проекте.

Листинг 13.21. Понижение всех миграций

```
dotnet ef database update 0
```

Указание 0 в качестве аргумента команды dotnet ef database update приводит к удалению всех миграций, которые были применены к БД. Это не то же самое, что и возвращение к первоначальной стартовой точке, поскольку БД DataAppDb и таблица __EFMigrationsHistory остаются существовать (хотя таблица пуста, т.к. миграции не применялись). Выполните команду из листинга 13.22, чтобы вернуться в совершенно чистое состояние.

Листинг 13.22. Сбрасывание БД

```
dotnet ef database drop --force
```

Команда полностью сбрасывает БД DataAppDb, включая таблицу __EFMigrations History. Если вы хотите, чтобы перед сбрасыванием БД запрашивалось подтверждение, тогда опустите аргумент --force.

Работа с множеством баз данных

Во всех примерах, рассмотренных в главе до сих пор, предполагалось, что проект работает только с одной БД. Используемые команды dotnet ef migrations и dotnet ef database инспектируют проект в поиске класса контекста, подключаются к БД с применением связанной строки подключения и выполняют свою работу.

Когда проект полагается на множество БД, что может быть связано с наличием одной БД для данных о товарах, а другой — для данных о пользователях, то класс контекста, на который воздействует операция миграции, должен указываться как часть командной строки. В последующих разделах в проект будет добавлена вторая БД и показано, каким образом создавать и применять к ней миграции.

Принятие решения о создании другой базы данных

База данных может содержать множество таблиц, каждая из которых способна хранить различные типы объектов данных. Во многих проектах это означает, что все данные, требуемые приложению, могут храниться в единственной БД и быть доступными через единственный класс контекста.

Наиболее распространенная причина присутствия в проекте множества БД связана с использованием стороннего пакета, работающего с Entity Framework Core, такого как ASP.NET Core Identity, который применяется для управления пользовательскими учетными записями, аутентификацией и авторизацией. Специальные данные приложения (эквивалент данных о товарах в примере приложения) хранятся в одной БД, а идентификационные данные — в другой БД. Потребность во множестве БД может также возникнуть, если проект должен работать с данными из унаследованного приложения или если к разным типам данных предъявляются отличающиеся требования, связанные с производительностью или безопасностью.

Расширение модели данных

Отправной точкой для добавления в пример приложения второй БД является создание нового сущностного класса, нового хранилища и класса его реализации. Начните с добавления в папку `Models` файла по имени `Customer.cs` и поместите в него код из листинга 13.23.

Листинг 13.23. Содержимое файла `Customer.cs` из папки `Models`

```
namespace DataApp.Models {

  public class Customer {

    public long Id { get; set; }
    public string Name { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
  }
}
```

В классе `Customer` определено свойство `Id`, которое будет использоваться в качестве уникального ключа; значения для ключа будут генерироваться БД. Свойства `Name`, `String` и `Country` представляют собой обычные строковые значения. Чтобы создать класс контекста для работы с объектами `Customer`, добавьте в папку `Models` файл по имени `EFCustomerContext.cs` с кодом, показанным в листинге 13.24.

Листинг 13.24. Содержимое файла `EFCustomerContext.cs` из папки `Models`

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

```
namespace DataApp.Models {
  public class EFCustomerContext : DbContext {

    public EFCustomerContext(DbContextOptions<EFCustomerContext> opts)
      : base(opts) { }

    public DbSet<Customer> Customers { get; set; }
  }
}
```

Класс `EFCustomerContext` следует стандартному шаблону для контекста: определяет конструктор, принимающий объект конфигурации, и свойство, возвращающее `DbSet<T>` с параметром типа, который указывает тип объектов, управляемых контекстом.

Чтобы создать хранилище для объектов `Customer`, добавьте в папку `Models` файл класса по имени `CustomerRepository.cs` и поместите в него код из листинга 13.25. Для простоты файл содержит и интерфейс, и класс реализации.

Листинг 13.25. Содержимое файла `CustomerRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {

  public interface ICustomerRepository {

    IEnumerable<Customer> GetAllCustomers();

  }

  public class EFCustomerRepository : ICustomerRepository {
    private EFCustomerContext context;

    public EFCustomerRepository(EFCustomerContext ctx) {
      context = ctx;
    }

    public IEnumerable<Customer> GetAllCustomers() {
      return context.Customers;
    }
  }
}
```

Хранилище предоставляет метод `GetAllCustomers()`, который возвращает все объекты `Customer` в БД. Ради максимального упрощения примера другие стандартные операции над данными, описанные в главе 12, здесь опущены.

Конфигурирование приложения

Строка подключения необходима для того, чтобы поставщик БД мог достичь сервера баз данных, аутентифицировать себя и использовать новую БД. Добавьте в файл `appsettings.json` строку подключения, как показано в листинге 13.26.

Листинг 13.26. Определение строки подключения в файле `appsettings.json` из папки `DataApp`

```
{
  "ConnectionStrings": {
```

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

Новая строка подключения `CustomerConnection` указывает БД по имени `CustomerDb` на том же сервере баз данных LocalDB, применяемом для хранения объектов `Product`.

Совет. Обе БД в примере приложения управляются одним сервером баз данных — SQL Server, доступ к которому осуществляется через средство LocalDB. Это не является обязательным требованием, и вы можете использовать в своем приложении отдельные серверы баз данных и даже смешивать серверы от разных поставщиков, скажем, SQL Server и MySQL.

Следующим шагом будет настройка инфраструктуры Entity Framework Core для взаимодействия с новой БД и конфигурирование средства внедрения зависимостей ASP.NET Core для обработки зависимостей от интерфейса `ICustomerRepository`. Добавьте в метод `ConfigureServices()` класса `Startup` операторы из листинга 13.27.

Листинг 13.27. Конфигурирование приложения в файле `Startup.cs` из папки `DataApp`

```
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 DataApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace DataApp {
  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<EFDatabaseContext>(options =>
        options.UseSqlServer(conString));
```

```
      string customerConString =
        Configuration["ConnectionStrings:CustomerConnection"];
      services.AddDbContext<EFCustomerContext>(options =>
        options.UseSqlServer(customerConString));
      services.AddTransient<IDataRepository, EFDataRepository>();
      services.AddTransient<ICustomerRepository, EFCustomerRepository>();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Создание и применение миграций

Теперь, когда в приложении присутствуют два класса контекста, используемые ранее в главе команды работать не будут. Чтобы увидеть проблему, попробуйте выполнить в папке DataApp команду, показанную в листинге 13.28.

Листинг 13.28. Попытка создания миграции

```
dotnet ef migrations add Customers_Initial
```

Отобразится сообщение об ошибке:

```
...
More than one DbContext was found. Specify which one to use.
Use the '-Context' parameter for PowerShell commands and the
'--context' parameter for dotnet commands.
Обнаружено более одного объекта DbContext. Укажите, какой из них использовать.
Применяйте параметр -Context в командах PowerShell
и параметр --context в командах dotnet.
...
```

При инспектировании проекта инфраструктура Entity Framework Core обнаруживает два класса, производных от DbContext, и не знает, к какому из них должна быть применена миграция.

Вам не придется просматривать проект в поисках обнаруженных классов контекста. Взамен выполните в папке DataApp команду, представленную в листинге 13.29, в результате чего инфраструктура Entity Framework Core отобразит список найденных классов контекста.

Листинг 13.29. Вывод списка классов контекста

```
dotnet ef dbcontext list
```

Команда отобразит список классов контекста, имеющихся в проекте:

```
...
DataApp.Models.EFCustomerContext
DataApp.Models.EFDatabaseContext
...
```

Имя класса вместе с пространством имен или без него можно использовать в аргументе `--context` для выбора контекста, с которым будет осуществляться работа. Выполните команды из листинга 13.30, находясь в папке `DataApp`, чтобы создать начальную миграцию для класса `Customer` и отразить неохваченные изменения в классе `Product`, который был создан после удаления миграций ранее в главе, на этот раз с указанием контекста для применения.

Листинг 13.30. Указание контекста при создании миграции

```
dotnet ef migrations add Initial --context EFCustomerContext
dotnet ef migrations add Current --context EFDatabaseContext
```

Такой же прием требуется для применения миграций к БД. Выполните в папке `DataApp` команды, приведенные в листинге 13.31, чтобы применить миграции к обеим БД.

Листинг 13.31. Применение миграций к БД

```
dotnet ef database update --context EFDatabaseContext
dotnet ef database update --context EFCustomerContext
```

Программное управление миграциями

В большинстве проектов лучший способ управления миграциями предусматривает использование инструментов командной строки `dotnet ef migrations` и `dotnet ef database`. Но инфраструктура Entity Framework Core также предлагает API-интерфейс для программного управления миграциями, который полезен в ситуациях, когда работа с командной строкой невозможна. Этот API-интерфейс позволяет создать инструмент управления миграциями, который может использоваться для принятия решений о том, какие миграции применять к БД. В настоящем разделе демонстрируется создание диспетчера миграций и работа API-интерфейса миграций.

Внимание! Не используйте инфраструктуру ASP.NET Core Identity для контроля доступа к инструменту управления миграциями. В плане хранения данных Identity полагается на Entity Framework Core, а это значит, что можно легко применить миграцию, которая предотвратит вашу аутентификацию и воспрепятствует использованию инструмента управления миграциями. Обычно я устанавливаю инструмент управления миграциями отдельно от сервера баз данных, изолируя его от общедоступных приложений MVC.

Создание класса диспетчера миграций

Отправной точкой при создании диспетчера миграций является определение вспомогательного класса, который позаботится о взаимодействии с API-интерфейсом Entity Framework Core способом, легким для потребления из контроллера MVC. Создайте в папке `Models` файл класса по имени `MigrationsManager.cs` и поместите в него содержимое листинга 13.32.

Листинг 13.32. Содержимое файла `MigrationsManager.cs` из папки `Models`

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace DataApp.Models {

  public class MigrationsManager {
    private IEnumerable<Type> ContextTypes;
    private IServiceProvider provider;
    public IEnumerable<string> ContextNames;

    public MigrationsManager(IServiceProvider prov) {
      provider = prov;

      ContextTypes = provider.GetServices<DbContextOptions>()
        .Select(o => o.ContextType);
      ContextNames = ContextTypes.Select(t => t.FullName);
      ContextName = ContextNames.First();
    }

    public string ContextName { get; set; }

    public IEnumerable<string> AppliedMigrations
      => Context.Database.GetAppliedMigrations();

    public IEnumerable<string> PendingMigrations
      => Context.Database.GetPendingMigrations();

    public IEnumerable<string> AllMigrations
      => Context.Database.GetMigrations();

    public void Migrate(string contextName, string target = null) {
      Context.GetService<IMigrator>().Migrate(target);
    }

    public DbContext Context =>
      provider.GetRequiredService(Type.GetType(ContextName)) as DbContext;
  }
}
```

Класс `MigrationsManager` исполняет три роли. Первая роль — извлечение типов и имен классов контекста БД в приложении. Класс диспетчера получает все объекты `DbContextOptions`, которые были зарегистрированы как службы с применением средства внедрения зависимостей ASP.NET Core, и затем, используя эти службы, получает ассоциированный класс контекста путем чтения свойства `ContextType` каждого объекта `DbContextOptions`:

```
...
ContextTypes =
  provider.GetServices<DbContextOptions>().Select(o => o.ContextType);
...
```

Подход отчасти неуклюж, но таким способом классы контекста обнаруживаются самой инфраструктурой Entity Framework Core.

Вторая роль класса диспетчера заключается в том, чтобы принять строку, которая содержит имя типа для класса контекста, и получить экземпляр данного класса с помощью средства внедрения зависимостей ASP.NET Core:

```
...
public DbContext Context =>
  provider.GetRequiredService(Type.GetType(ContextName)) as DbContext;
...
```

Это связка с частью MVC приложения, получающей имя контекста, к которому должна применяться миграция, как строку в HTML-форме и использующей ее для установки значения свойства `ContextName`, которое применяется для идентификации желаемого пользователем контекста.

Третья роль класса диспетчера — выполнение объектов миграции на контекстах БД. Пространство имен `Microsoft.EntityFrameworkCore` содержит описанные в табл. 13.8 расширяющие методы, которые оперируют на миграциях через объекты `DatabaseFacade`, возвращаемые свойством `DbContext.Database`.

Таблица 13.8. Расширяющие методы `DatabaseFacade`, предназначенные для миграций

Имя	Описание
`GetMigrations()`	Возвращает последовательность строковых значений, каждое из которых является именем миграции
`GetAppliedMigrations()`	Возвращает последовательность строковых значений, каждое из которых является именем миграции, примененной к БД
`GetPendingMigrations()`	Возвращает последовательность строковых значений, каждое из которых является именем миграции, пока еще не примененной к БД
`Migrate()`	Применяет все ожидающие миграции к БД

Метод `Migrate()` не позволяет указывать специфическую миграцию. Предоставление такой функциональности означает использование службы `IMigrator` из пространства имен `Microsoft.EntityFrameworkCore.Migrations`, которая определяет метод `Migrate()`, позволяющий указывать миграцию. Именно он задействован в реализации метода `Migrate()` класса `MigrationsManager`:

```
...
Context.GetService<IMigrator>().Migrate(target);
...
```

За счет применения методов, описанных в табл. 13.8, и службы `IMigrator` класс диспетчера способен снабжать информацией о состоянии каждого класса контекста БД, а также применять и удалять индивидуальные миграции.

Создание контроллера и представления для диспетчера миграций

Следующим шагом будет создание контроллера и представления, которые предоставят доступ к функциональности, предлагаемой классом `MigrationsManager`. Добавьте в папку `Controllers` файл класса по имени `MigrationsController.cs` с содержимым, показанным в листинге 13.33.

Листинг 13.33. Содержимое файла `MigrationsController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {

  public class MigrationsController : Controller {
    private MigrationsManager manager;

    public MigrationsController(MigrationsManager mgr) {
      manager = mgr;
    }

    public IActionResult Index(string context) {
      ViewBag.Context = manager.ContextName = context
        ?? manager.ContextNames.First();
      return View(manager);
    }

    [HttpPost]
    public IActionResult Migrate(string context, string migration) {
      manager.ContextName = context;
      manager.Migrate(context, migration);
      return RedirectToAction(nameof(Index), new { context = context });
    }
  }
}
```

Метод действия `Index()` используется для отображения деталей классов контекста и миграций в проекте. Метод `Migrate()`, принимающий только запросы `POST`, служит для применения миграций к БД. Чтобы снабдить контроллер представлением, создайте папку `Views/Migrations`, добавьте в нее файл Razor по имени `Index.cshtml` и поместите в него разметку из листинга 13.34.

Листинг 13.34. Содержимое файла `Index.cshtml` из папки `Views/Migrations`

```
@using DataApp.Models
@model MigrationsManager
@{
  ViewData["Title"] = "Migrations";
  Layout = "_Layout";
}

<div class="m-1 p-2">
  <form asp-action="Index" method="get" class="form-inline">
    <label class="m-1">Database Context:</label>
    <select name="context" class="form-control">
      @foreach (var name in Model.ContextNames) {
        <option selected="@(name == ViewBag.Context)">@name</option>
      }
    </select>
    <button class="btn btn-primary m-1">Select</button>
  </form>
</div>
```

```
<table class="table table-sm table-striped m-2">
  <thead>
    <tr><th>Migration Name</th><th>Status</th></tr>
  </thead>
  <tbody>
    @foreach (string m in Model.AllMigrations) {
      <tr>
        <td>@m</td>
        <td>
          @(Model.AppliedMigrations.Contains(m)
            ? "Applied" : "Pending")
        </td>
      </tr>
    }
  </tbody>
</table>

<div class="m-1 p-2">
  <form asp-action="Migrate" method="post" class="form-inline">
    <input type="hidden" name="context" value="@ViewBag.Context" />
    <label class="m-1">Migration:</label>
    <select name="migration" class="form-control">
      <option selected value="@Model.AllMigrations.Last()">All</option>
      @foreach (var m in Model.AllMigrations.Reverse()) {
        <option>@m</option>
      }
      <option value="0">None</option>
    </select>
    <button class="btn btn-primary m-1">Migrate</button>
  </form>
</div>
```

Представление содержит форму, которая используется для выбора управляемого класса контекста, таблицу со списком миграций и их состоянием и еще одну форму, предназначенную для применения миграций.

Конфигурирование приложения

Класс `MigrationsManager` должен быть зарегистрирован в системе внедрения зависимостей ASP.NET Core, чтобы новые экземпляры можно было создавать с помощью объекта реализации `IServiceProvider`, который используется для получения объектов контекста. Добавьте в метод `ConfigureServices()` класса `Startup` оператор, приведенный в листинге 13.35.

Листинг 13.35. Регистрация вспомогательного класса в файле `Startup.cs` из папки `DataApp`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddMvc();
  string conString = Configuration["ConnectionStrings:DefaultConnection"];
  services.AddDbContext<EFDatabaseContext>(options =>
    options.UseSqlServer(conString));
```

```
  string customerConString =
    Configuration["ConnectionStrings:CustomerConnection"];
  services.AddDbContext<EFCustomerContext>(options =>
    options.UseSqlServer(customerConString));

  services.AddTransient<IDataRepository, EFDataRepository>();
  services.AddTransient<ICustomerRepository, EFCustomerRepository>();
  services.AddTransient<MigrationsManager>();
}
...
```

Класс `MigrationsManager` регистрируется с применением метода `AddTransient()`, т.е. каждая зависимость от класса распознается посредством нового объекта.

Выполнение диспетчера миграций

Перезапустите приложение, используя команду `dotnet run`, и перейдите по ссылке `http://localhost:5000/migrations`. Вы можете переключаться между классами контекста, выбирая их с помощью элемента `select` в верхней части окна и щелкая на кнопке Select (Выбрать). Для повышения или понижения БД выберите желаемую миграцию и щелкните на кнопке Migrate (Выполнить миграцию), как показано на рис. 13.2. Просмотрите вывод из приложения; вы увидите SQL-операторы, которые отправляются серверу баз данных для перемещения между миграциями.

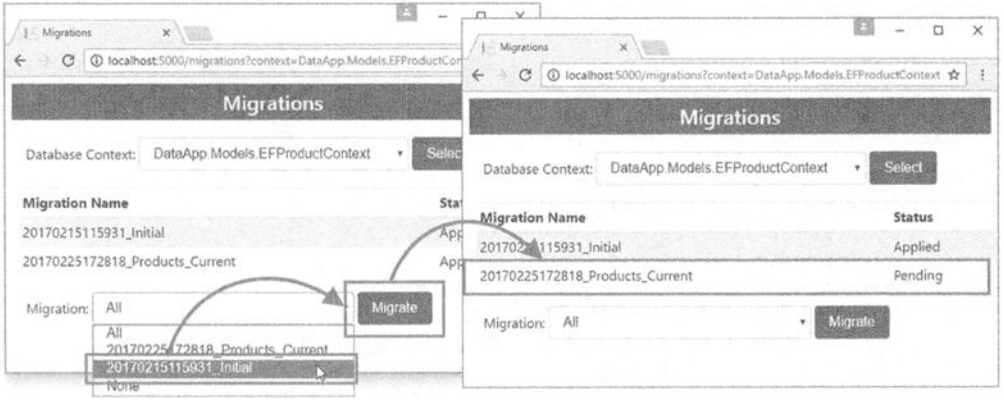

Рис. 13.2. Программное управление миграциями

Программное заполнение баз данных начальными данными

Многие БД нуждаются в начальных данных, чтобы приложение имело некоторые исходные данные для работы. Подобная потребность часто возникает с БД для товаров, где имеется первоначальный набор товаров для продажи клиентам, а также с БД для безопасности, где есть, по меньшей мере, одна учетная запись, позволяющая администраторам войти и выполнить исходные задачи конфигурирования.

До сих пор в этой части книги БД заполнялись начальными данными с применением низкоуровневого кода SQL, посылаемого прямо серверу баз данных. Такой способ добавления данных в БД чреват ошибками и не использует Entity Framework Core.

Инфраструктура Entity Framework Core не имеет встроенной поддержки для работы с начальными данными, хотя на время написания главы она упоминалась в опубликованной дорожной карте. До тех пор, пока не появится специальное средство начального заполнения, можно применять два приема, которые описаны в последующих разделах. Каждый подход обладает преимуществами и недостатками, но общим эффектом будет заполнение БД, так что использовать низкоуровневый код SQL не придется.

Оба подхода основаны на создании объектов и их сохранении в БД с применением Entity Framework Core. Чтобы избежать повторения кода для каждого подхода, создайте в папке `Models` файл класса по имени `SeedData.cs` и поместите в него код из листинга 13.36.

Внимание! Код в листинге 13.36 содержит метод, который удаляет все данные из БД. Он удобен при экспериментировании с Entity Framework Core, поскольку позволяет легко сбросить БД и опробовать другой прием начального заполнения. В реальных проектах необходимо соблюдать осторожность, чтобы не удалить производственные данные, и вы можете уменьшить риск, не реализуя эквивалент метода `ClearData()` в собственных проектах.

Листинг 13.36. Содержимое файла `SeedData.cs` из папки `Models`

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

namespace DataApp.Models {

  public static class SeedData {

    public static void Seed(DbContext context) {
      if (context.Database.GetPendingMigrations().Count() == 0) {
        if (context is EFDatabaseContext prodCtx
            && prodCtx.Products.Count() == 0) {
          prodCtx.Products.AddRange(Products);
        } else if (context is EFCustomerContext custCtx
                && custCtx.Customers.Count() == 0) {
          custCtx.Customers.AddRange(Customers);
        }
        context.SaveChanges();
      }
    }

    public static void ClearData(DbContext context) {
      if (context is EFDatabaseContext prodCtx
          && prodCtx.Products.Count() > 0) {
        prodCtx.Products.RemoveRange(prodCtx.Products);
      } else if (context is EFCustomerContext custCtx
              && custCtx.Customers.Count() > 0) {
        custCtx.Customers.RemoveRange(custCtx.Customers);
      }
      context.SaveChanges();
    }
```

```
    private static Product[] Products = {
      new Product { Name = "Kayak", Category = "Watersports",
        Price = 275, Color = Colors.Green, InStock = true },
      new Product { Name = "Lifejacket", Category = "Watersports",
        Price = 48.95m, Color = Colors.Red, InStock = true },
      new Product { Name = "Soccer Ball", Category = "Soccer",
        Price = 19.50m, Color = Colors.Blue, InStock = true },
      new Product { Name = "Corner Flags", Category = "Soccer",
        Price = 34.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Stadium", Category = "Soccer",
        Price = 79500, Color = Colors.Red, InStock = true },
      new Product { Name = "Thinking Cap", Category = "Chess",
        Price = 16, Color = Colors.Blue, InStock = true },
      new Product { Name = "Unsteady Chair", Category = "Chess",
        Price = 29.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Human Chess Board", Category = "Chess",
        Price = 75, Color = Colors.Red, InStock = true },
      new Product { Name = "Bling-Bling King", Category = "Chess",
        Price = 1200, Color = Colors.Blue, InStock = true });

    private static Customer[] Customers = {
      new Customer { Name = "Alice Smith",
        City = "New York", Country = "USA" },
      new Customer { Name = "Bob Jones",
        City = "Paris", Country = "France" },
      new Customer { Name = "Charlie Davies",
        City = "London", Country = "UK" });
  }
}
```

В классе `SeedData` определены статические свойства, создающие объекты `Product` и `Customer` для использования в качестве начальных данных, и методы, которые применяют статические свойства для заполнения и очистки БД. Каждый метод принимает объект контекста БД, идентифицирует тип контекста и затем добавляет или удаляет данные.

Метод `Seed()`, который отвечает за добавление начальных данных в БД, первым делом проверяет, не остались ли ожидающие миграции:

```
...
if (context.Database.GetPendingMigrations().Count() == 0) {
...
```

Программное заполнение БД начальными данными может быть сложным, потому что объекты, используемые для заполнения БД, должны соответствовать структуре таблицы БД, которую Entity Framework Core будет применять для их хранения. В случае несовпадения инфраструктура Entity Framework Core будет неспособна сохранить данные, и начальное заполнение потерпит неудачу. Проверка отсутствия ожидающих миграций перед попыткой сохранения начальных данных помогает снизить риск несовпадения, но вы все равно должны позаботиться о том, чтобы все изменения в классах модели данных были зафиксированы в процессе миграции, прежде чем пытаться заполнить БД начальными данными.

Создание инструмента заполнения начальными данными

Первый подход предусматривает создание инструмента заполнения БД начальными данными, похожего на тот, что создавался ранее в главе для управления миграциями. На самом деле во избежание дублирования кода мы расширим инструмент управления миграциями, чтобы он также мог заполнять БД начальными данными.

Это лучший подход к заполнению БД начальными данными в производственных системах. Его недостаток в том, что заполнение начальными данными требует от разработчика или администратора явного выполнения действия по добавлению начальных данных в БД, которое может утомлять на этапе разработки (здесь больше подходит прием, описанный в разделе "Заполнение начальными данными во время запуска" далее в главе).

В качестве первого шага добавьте в контроллер `Migrations` новые методы действий, которые могут использоваться для заполнения начальными данными и очистки БД (листинг 13.37).

Листинг 13.37. Добавление методов действий в файле `MigrationsController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class MigrationsController : Controller {
    private MigrationsManager manager;
    public MigrationsController(MigrationsManager mgr) {
      manager = mgr;
    }
    public IActionResult Index(string context) {
      ViewBag.Context = manager.ContextName = context
        ?? manager.ContextNames.First();
      return View(manager);
    }
    [HttpPost]
    public IActionResult Migrate(string context, string migration) {
      manager.ContextName = context;
      manager.Migrate(context, migration);
      return RedirectToAction(nameof(Index), new { context = context });
    }
    [HttpPost]
    public IActionResult Seed(string context) {
      manager.ContextName = context;
      SeedData.Seed(manager.Context);
      return RedirectToAction(nameof(Index), new { context = context });
    }
    [HttpPost]
    public IActionResult Clear(string context) {
      manager.ContextName = context;
      SeedData.ClearData(manager.Context);
      return RedirectToAction(nameof(Index), new { context = context });
    }
  }
}
```

На новые методы действий `Seed()` и `Clear()` можно нацеливаться только с применением запроса `POST`. Каждый метод принимает в своем аргументе имя класса контекста БД, используемого с созданным в листинге 13.32 классом `MigrationsManager` для получения объекта контекста, который может быть передан одному из методов, определенных в классе `SeedData`.

Располагая методами действий, можно создать кнопки, щелчки на которых будут инициировать заполнение начальными данными и очистку БД (листинг 13.38).

Листинг 13.38. Добавление кнопок в файле `Index.chstml` из папки `Views/Migrations`

```
@using DataApp.Models
@model MigrationsManager
@{
  ViewData["Title"] = "Migrations";
  Layout = "_Layout";
}
<!-- ...для краткости остальные элементы не показаны... -->
<div class="m-1 p-2">
  <form method="post">
    <input type="hidden" name="context" value="@ViewBag.Context" />
    <button class="btn btn-primary" asp-action="Seed">Seed Database
    </button>
    <button class="btn btn-danger" asp-action="Clear">Clear Database
    </button>
  </form>
</div>
```

Новые элементы определяют форму с действиями, указанными каждым элементом `button` через атрибут вспомогательной функции дескриптора `asp-action`. Чтобы увидеть результат, запустите приложение с применением команды `dotnet run`, перейдите по ссылке `http://localhost:5000/migrations` и щелкните на кнопке Seed Database (Заполнить БД начальными данными) или Clear Database (Очистить БД), как показано на рис. 13.3.

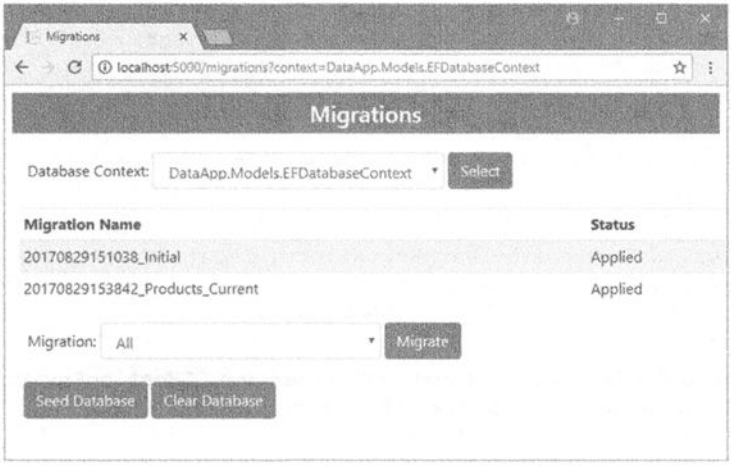

Рис. 13.3. Добавление к инструменту управления миграциями поддержки заполнения БД начальными данными

Заполнение начальными данными во время запуска

Еще один способ заполнения БД начальными данными предполагает делать это автоматически во время запуска приложения. Так следует поступать только на стадии разработки, особенно если в производственной среде планируется функционирование множества экземпляров приложения; в противном случае несколько экземпляров приложения могут попытаться заполнить БД начальными данными одновременно, вызывая проблемы и препятствуя чистому запуску.

Преимущество этого подхода заключается в том, что он полностью автоматический. Если БД не имеет ожидающих миграций и пуста, тогда она будет заполняться начальными данными. Чтобы заполнить БД начальными данными при запуске приложения, добавьте в класс `Startup` операторы, представленные в листинге 13.39.

Листинг 13.39. Заполнение БД начальными данными в файле `Startup.cs` из папки `DataApp`

```
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 DataApp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace DataApp {
  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<EFDatabaseContext>(options =>
        options.UseSqlServer(conString));

      string customerConString =
        Configuration["ConnectionStrings:CustomerConnection"];
      services.AddDbContext<EFCustomerContext>(options =>
        options.UseSqlServer(customerConString));

      services.AddTransient<IDataRepository, EFDataRepository>();
      services.AddTransient<ICustomerRepository, EFCustomerRepository>();
      services.AddTransient<MigrationsManager>();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env,
        EFDatabaseContext prodCtx, EFCustomerContext custCtx) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
```

```
      if (env.IsDevelopment()) {
        SeedData.Seed(prodCtx);
        SeedData.Seed(custCtx);
      }
    }
  }
}
```

Объявление параметров метода `Configure()` позволяет получить объекты контекста для БД, которые могут быть переданы методу `SeedData.Seed()`. Чтобы предотвратить заполнение БД начальными данными в производственной среде, производится проверка среды на предмет того, что приложение функционирует в среде разработки. В результате БД заполняется начальными данными автоматически при запуске приложения, в чем можно удостовериться, очистив БД с помощью инструмента из предыдущего раздела и перезапустив приложение.

Избегание проблем с инструментами командной строки

В выпуске Entity Framework Core 2 разработчики из компании Microsoft изменили способ обнаружения контекстов БД, а это означает, что код, добавленный в листинге 13.39, будет выполняться, даже если для управления миграциями или применения их к БД используются инструменты командной строки. В результате может возникнуть ситуация, когда код заполнения начальными данными пытается записать в БД данные, которые она не способна сохранить или которые не существуют. Если подобное произошло, тогда при запуске команды `dotnet ef` вы увидите исключение. Чтобы решить проблему, поместите в комментарии вызовы метода `SeedData.Seed()`, добавленные в листинге 13.39, на время работы с миграции и по завершении удалите комментарии.

Резюме

В главе объяснялось, как применять миграции для поддержания схемы БД в согласованном с приложением состоянии и проводить подготовку БД, чтобы они могли хранить данные приложения. Был рассмотрен процесс создания новых миграций, показано, каким образом повышать и понижать БД, и продемонстрированы способы применения и управления миграциями с использованием API-интерфейса Entity Framework Core. В заключение в главе было показано, как заполнять БД начальными данными для производственной среды и среды разработки. Следующая глава посвящена созданию отношений между объектами и представлению их в БД.

ГЛАВА 14

Создание отношений между данными

Основой инфраструктуры Entity Framework Core является способ, которым она представляет экземпляры классов .NET в виде строк внутри таблицы реляционной БД. На создание отношений между классами Entity Framework Core реагирует созданием соответствующих отношений в БД. В целом процесс создания отношений между данными интуитивно понятен и естественен, хотя доступно много вариантов, а некоторые более сложные возможности скрывают в себе ловушки для неосторожных.

В этой главе будет продемонстрировано, как создавать отношения между классами в модели данных, показано, каким образом Entity Framework Core реагирует на такие изменения, и объяснено, как использовать отношения в приложении ASP.NET Core MVC. В табл. 14.1 приведены сведения, позволяющие поместить отношения в контекст.

Таблица 14.1. Помещение отношений в контекст

Вопрос	Ответ
Что это такое?	Отношения позволяют БД хранить ассоциации между объектами в модели данных
Чем они полезны?	Отношения дают возможность более естественно моделировать данные с применением объектов .NET и затем сохранять эти объекты и их ассоциации с другими объектами
Как они используются?	Отношения создаются путем добавления свойств в классы модели данных с последующим использованием миграции для обновления БД. При сохранении данных инфраструктура Entity Framework Core будет пытаться сохранить связанные данные автоматически, но не всегда работает так, как можно было ожидать
Существуют ли какие-то скрытые ловушки или ограничения?	Отношениям присуща сложность, поскольку способ, которым БД управляют ассоциациями между данными, не всегда отражает естественное поведение объектов .NET. Получение требуемого поведения может быть сопряжено с рядом усилий
Существуют ли альтернативы?	Вы не обязаны применять отношения вообще и при желании можете просто хранить индивидуальные объекты, но такой подход будет ограничивать приложения только простыми моделями данных

В табл. 14.2 приведена сводка по главе.

Таблица 14.2. Сводка по главе

Задача	Решение	Листинг
Создание отношения	Добавьте навигационное свойство, после чего создайте и примените миграцию	14.1–14.4, 14.10
Включение связанных данных в запрос	Используйте методы `Include()` и `ThenInclude()`	14.5–14.9, 14.22–14.29
Сохранение или обновление связанных данных	Применяйте методы, предоставляемые классом контекста	14.12–14.15
Удаление связанных данных	Используйте метод, предоставляемый классом контекста	14.16
Создание обязательного отношения	Добавьте свойство внешнего ключа с типом, не допускающим значения `null`	14.17–14.21

Подготовительные шаги

В главе продолжается работа с проектом `DataApp`, созданным в главе 11 и модифицированным в последующих главах. В качестве подготовки откройте окно командной строки, перейдите в папку `DataApp` и выполните команду из листинга 14.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 14.1. Удаление БД

```
dotnet ef database drop --force --context EFDatabaseContext
```

Команда удаляет БД, применяемую для хранения объектов `Product`, что поможет вам получить ожидаемые результаты в последующих примерах. Приложение не будет располагать БД или данными для работы до демонстрации процесса создания отношения между данными позже в главе.

Создание отношения

Лучший способ понять суть отношений между данными — создать одно такое отношение. Первым делом понадобится добавить в приложение новый сущностный класс для представления поставщика товара. Добавьте в папку `Models` файл класса по имени `Supplier.cs` и поместите в него код, показанный в листинге 14.2.

Листинг 14.2. Содержимое файла `Supplier.cs` из папки `Models`

```
namespace DataApp.Models {

  public class Supplier {
    public long Id { get; set; }
    public string Name { get; set; }
```

```
    public string City { get; set; }
    public string State { get; set; }
  }
}
```

В новом классе определено свойство `Id`, которое будет использоваться для хранения значений первичного ключа в БД, а также свойства `Name`, `City` и `State`, предоставляющие основное описание поставщика товара. В настоящий момент с классом `Supplier` не связано ничего особенного; он следует тому же шаблону, который применялся в предшествующих примерах.

Добавление навигационного свойства

Чтобы создать отношение между данными, добавьте в класс `Product` свойство, которое позволит каждому объекту `Product` быть ассоциированным с объектом `Supplier` (листинг 14.3).

Листинг 14.3. Добавление свойства в файле `Product.cs` из папки `Models`

```
namespace DataApp.Models {

  public enum Colors {
    Red, Green, Blue
  }

  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    public Colors Color { get; set; }
    public bool InStock { get; set; }

    public Supplier Supplier { get; set; }
  }
}
```

Новое свойство по имени `Supplier` создает отношение, так что каждый объект `Product` может быть ассоциирован с одним объектом `Supplier`. Оно называется *навигационным свойством*, поскольку делает возможной навигацию с одного объекта на другой. В рассматриваемом случае навигационное свойство в объекте `Product` можно использовать для получения доступа к связанному объекту `Supplier`.

Именование навигационных свойств

В листинге 14.3 навигационному свойству, которое создает отношение с объектом `Supplier`, было назначено имя `Supplier`. Выбирать для навигационного свойства имя связанного класса вовсе не обязательно, хотя поступать так обычно удобно, что и было сделано в данном случае. Можно применять любое допустимое имя для свойства C#; позже в главе будет приведен пример, в котором свойство, создающее отношение с экземпляром класса `ContactLocation`, получает имя `Location`.

Для лучшего понимания особенностей обработки отношений инфраструктурой Entity Framework Core полезно подумать о том, каким образом свойство `Supplier` влияет на объекты .NET в приложении, не беспокоясь пока о БД. Внесенные в модель данных изменения означают, что каждый объект `Product` может быть связан с объектом `Supplier` (рис. 14.1).

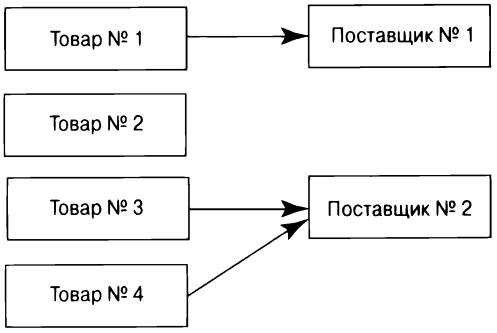

Рис. 14.1. Создание отношения между классами модели данных

Свойство `Supplier` объекта `Product` способно иметь значение `null`, указывающее на то, что объект `Product` не должен быть связан с каким-либо объектом `Supplier`. Кроме того, свойство `Supplier` объекта `Product` не обязано быть уникальным, т.е. несколько объектов `Product` могут быть связаны с одним и тем же объектом `Supplier`.

Создание миграции

Чтобы посмотреть, как Entity Framework Core обрабатывает добавление навигационного свойства, выполните в папке проекта `DataApp` команду из листинга 14.4 для создания миграции.

Листинг 14.4. Создание миграции для обновления БД

```
dotnet ef migrations add Add_Supplier --context EFDatabaseContext
```

Инфраструктура Entity Framework Core создаст миграцию по имени `Add_Supplier`. Заглянув в метод `Up()` внутри файла `<отметка времени>_Add_Supplier.cs` из папки `Migrations`, вы увидите, каким образом Entity Framework Core будет обновлять БД. Метод `Up()` содержит четыре оператора, которые обновят БД. Первый оператор добавляет новый столбец в существующую таблицу БД:

```
...
migrationBuilder.AddColumn<long>(name: "SupplierId",
                                 table: "Products",
                              nullable: true);
...
```

Этот столбец инфраструктура Entity Framework Core будет использовать для отслеживания объекта `Supplier`, ассоциированного с объектом `Product`. Аргументы метода `AddColumn()` приведут к созданию столбца по имени `SupplierId` в существующей таблице `Products`. Аргумент `nullable` установлен в `true`, что указывает

на допустимость наличия в новом столбце значений null, позволяя хранить в БД объекты Product, которые не связаны с какими-либо объектами Supplier.

Второй оператор в методе Up() миграции создает новую таблицу, которая будет применяться для хранения объектов Supplier:

```
...
migrationBuilder.CreateTable(
  name: "Supplier",
  columns: table => new {
    Id = table.Column<long>(nullable: false)
      .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn),
    City = table.Column<string>(nullable: true),
    Name = table.Column<string>(nullable: true),
    State = table.Column<string>(nullable: true)
  },
  constraints: table => {
    table.PrimaryKey("PK_Supplier", x => x.Id);
  });
...
```

Аргументы метода CreateTable() обеспечат создание таблицы по имени Supplier со столбцами Id, Name, City и State, которые соответствуют свойствам, определенным в классе Supplier. Столбец Id сконфигурирован как первичный ключ, и сервер баз данных будет ответственным за генерирование значений для этого столбца при добавлении новых строк в таблицу.

Совет. Обратите внимание, что именем таблицы является Supplier, тогда как существующая таблица имеет имя Products. По соглашению инфраструктура Entity Framework Core использует имя свойства DbSet<T>, когда создает таблицу для сущностного класса, доступного через класс контекста, и имя навигационного свойства, когда создает таблицу для сущностного класса, который может быть доступен только через отношение.

Следующий оператор создает индекс, позволяющий ускорить выполнение запросов к БД:

```
...
migrationBuilder.CreateIndex(name: "IX_Products_SupplierId",
  table: "Products", column: "SupplierId");
...
```

Индекс не связан напрямую с представлением отношения между объектами Product и Supplier, а потому в настоящей главе он не обсуждается. Последний оператор в методе Up() несет ответственность за создание связи между строками в таблицах Products и Supplier:

```
...
migrationBuilder.AddForeignKey(
  name: "FK_Products_Supplier_SupplierId", table: "Products",
  column: "SupplierId", principalTable: "Supplier",
  principalColumn: "Id", onDelete: ReferentialAction.Restrict);
...
```

Метод AddForeignKey() применяется для конфигурирования столбца SupplierId, добавленного в таблицу Products, чтобы создать отношение внешнего

ключа со столбцом Id новой таблицы Supplier. Отношение внешнего ключа помогает защитить целостность БД, поскольку гарантирует, что строка таблицы Product может иметь в столбце SupplierId только такое значение, которое соответствует допустимой строке в таблице Supplier (или null, указывая на отсутствие связанного объекта Supplier).

Совокупный эффект операторов в методе Up() миграции — обновление БД с целью отражения изменений, внесенных в модель данных (рис. 14.2).

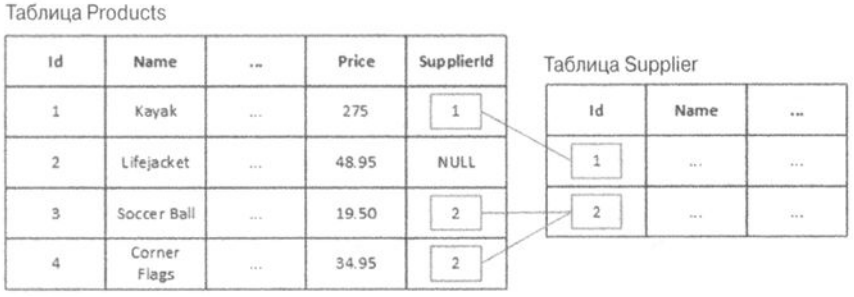

Рис. 14.2. Отношение между данными в БД

Таблица Supplier позволяет инфраструктуре Entity Framework Core сохранять объекты Supplier, а столбец SupplierId, добавленный в таблицу Products, будет использоваться для отслеживания связей между объектами Product и Supplier.

Запрашивание и отображение связанных данных

В главе объясняется, как выполнять полный набор операций над связанными данными, но имеет смысл начать с самой элементарной задачи — отображение связанных значений данных для пользователя. По умолчанию инфраструктура Entity Framework Core не следует по навигационным свойствам с целью загрузки связанных данных, чтобы предотвратить возвращение запросами данных, которые не требуются. Для загрузки данных Supplier, связанных с объектом Product, добавьте в класс реализации хранилища операторы, показанные в листинге 14.5.

Листинг 14.5. Запрашивание связанных данных в файле EFDataRepository.cs из папки Models

```
using System;
using System.Collections.Generic;
using System.Linq;
using Newtonsoft.Json;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore;

namespace DataApp.Models {

  public class EFDataRepository : IDataRepository {
    private EFDatabaseContext context;

    public EFDataRepository(EFDatabaseContext ctx) {
      context = ctx;
    }
```

```
    public Product GetProduct(long id) {
      return context.Products.Include(p => p.Supplier).First(p => p.Id == id);
    }
    public IEnumerable<Product> GetAllProducts() {
      Console.WriteLine("GetAllProducts");
      return context.Products.Include(p => p.Supplier);
    }
    public IEnumerable<Product> GetFilteredProducts(string category = null,
        decimal? price = null, bool includeRelated = true) {
      IQueryable<Product> data = context.Products;
      if (category != null) {
        data = data.Where(p => p.Category == category);
      }
      if (price != null) {
        data = data.Where(p => p.Price >= price);
      }
      if (includeRelated) {
        data = data.Include(p => p.Supplier);
      }
      return data;
    }
    public void CreateProduct(Product newProduct) {
      newProduct.Id = 0;
      context.Products.Add(newProduct);
      context.SaveChanges();
      Console.WriteLine($"New Key: {newProduct.Id}");
    }
    public void UpdateProduct(Product changedProduct,
        Product originalProduct = null) {
      if (originalProduct == null) {
        originalProduct = context.Products.Find(changedProduct.Id);
      } else {
        context.Products.Attach(originalProduct);
      }
      originalProduct.Name = changedProduct.Name;
      originalProduct.Category = changedProduct.Category;
      originalProduct.Price = changedProduct.Price;
      context.SaveChanges();
    }
    public void DeleteProduct(long id) {
      Product p = context.Products.Find(id);
      context.Products.Remove(p);
      context.SaveChanges();
    }
  }
}
```

Расширяющий метод `Include()` определен в пространстве имен `Microsoft.EntityFrameworkCore` и вызывается на объектах, реализующих интерфейс `IQueryable<T>`, для включения связанных данных.

Аргументом метода `Include()` является выражение, выбирающее навигационное свойство, по которому инфраструктура Entity Framework Core должна проследовать, чтобы получить связанные данные. В листинге 14.5 выбирается объект `Supplier`. Метод `Include()` можно встраивать в запросы подобно другим методам LINQ, например:

```
...
return context.Products.Include(p => p.Supplier);
...
```

Применять на объекте, который возвращается методом `Include()`, допускается далеко не все расширяющие методы LINQ и потому в случае использования связанных данных может возникнуть необходимость в переделке ряда запросов. Метод `Find()`, применяемый в предшествующих примерах для нахождения специфического объекта `Product` в БД, нельзя использовать с методом `Include()`, поэтому он заменен методом `First()`, который работает аналогичным образом:

```
...
return context.Products.Include(p => p.Supplier).First(p => p.Id == id);
...
```

Вы можете принять решение о включении связанных данных в запрос во время выполнения. В листинге 14.5 метод `GetFilteredProducts()` изменен так, что метод `Include()` вызывается, только когда значение параметра `includeRelated` равно `true`:

```
...
if (includeRelated) {
  data = data.Include(p => p.Supplier);
}
...
```

Чтобы обеспечить поддержку избирательного включения связанных данных в остальных частях приложения, измените сигнатуру метода в интерфейсе `IDataRepository`, как показано в листинге 14.6.

Будущая поддержка ленивой загрузки

Ранние версии Entity Framework поддерживали средство, называемое *ленивой загрузкой*, которое по заявлениям Microsoft будет включено в будущий выпуск Entity Framework Core. При ленивой загрузке связанные данные загружаются из БД автоматически, когда читается навигационное свойство. В примере приложения это означало бы, что чтение значения свойства `Supplier` объекта `Product` автоматически инициировало бы SQL-запрос данных, требующихся для создания объекта `Supplier`.

Ленивая загрузка выглядит удачной идеей и часто делается доступной как удобная функция, которая позволяет разрабатывать часть MVC приложения без необходимости в создании запросов Entity Framework Core, включающих или исключающих разные наборы связанных данных для разных действий. Все работает словно по волшебству, и Entity Framework Core обеспечивает доступность данных, нужных части MVC приложения. Но "за кулисами" при чтении навигационных свойств генерируются дополнительные SQL-запросы, повышая нагрузку на сервер баз данных и увеличивая время обработки HTTP-запросов.

Я рекомендую избегать применения ленивой загрузки. Взамен определите в хранилище методы или свойства, которые возвращают данные во всех комбинациях, требуемых частью MVC приложения, со связанными данными или без них. Иными словами, избегайте использования любых средств, которые генерируют запросы к БД автоматически.

Листинг 14.6. Добавление параметра в файле `IDataRepository.cs` из папки `Models`

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

namespace DataApp.Models {

  public interface IDataRepository {

    Product GetProduct(long id);

    IEnumerable<Product> GetAllProducts();

    IEnumerable<Product> GetFilteredProducts(string category = null,
      decimal? price = null, bool includeRelated = true);

    void CreateProduct(Product newProduct);

    void UpdateProduct(Product changedProduct,
      Product originalProduct = null);

    void DeleteProduct(long id);
  }
}
```

Далее добавьте параметр к методу действия, чтобы контроллер мог получать значение, для которого нужно включить связанные данные, из HTTP-запроса через процесс привязки моделей MVC (листинг 14.7).

Листинг 14.7. Добавление параметра к методу действия в файле `HomeController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {
  public class HomeController : Controller {
    private IDataRepository repository;

    public HomeController(IDataRepository repo) {
      repository = repo;
    }

    public IActionResult Index(string category = null,
        decimal? price = null, bool includeRelated = true) {
      var products = repository
        .GetFilteredProducts(category, price, includeRelated);
      ViewBag.category = category;
      ViewBag.price = price;
      ViewBag.includeRelated = includeRelated;
      return View(products);
    }

    // ...для краткости остальные действия не показаны...
  }
}
```

Недостаток способа, которым инфраструктура Entity Framework Core обрабатывает связанные данные, обусловлен отсутствием эффективного подхода к выяснению,

равно ли навигационное свойство значению null. Навигационное свойство равно null либо из-за того, что для специфического объекта нет связанных данных, либо потому, что связанные данные не были выбраны с применением метода Include(). Для обхода проблемы использовалось свойство ViewBag, поэтому представлению будет известно, запрашивались ли связанные данные.

Обновление представления с целью отображения связанных данных

Теперь, когда в код внесены все необходимые изменения, связанные данные можно отображать с применением представления Razor. Модель данных будет включать объекты Supplier, когда они были запрошены, а свойство объекта ViewBag можно использовать для конфигурирования элемента form (листинг 14.8).

Листинг 14.8. Отображение связанных данных в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<DataApp.Models.Product>
@{
  ViewData["Title"] = "Products";
  Layout = "_Layout";
}
<div class="m-1 p-2">
  <form asp-action="Index" method="get" class="form-inline">
    <label class="m-1">Category:</label>
    <select name="category" class="form-control">
      <option value="">All</option>
      <option selected="@(ViewBag.category == "Watersports")">
        Watersports
      </option>
      <option selected="@(ViewBag.category == "Soccer")">Soccer</option>
      <option selected="@(ViewBag.category == "Chess")">Chess</option>
    </select>
    <label class="m-1">Min Price:</label>
    <input class="form-control" name="price" value="@ViewBag.price" />
    <div class="form-check m-1">
      <label class="form-check-label">
        <input class="form-check-input" type="checkbox"
              name="includeRelated" value="true"
              checked="@(ViewBag.includeRelated == true)"/>
          Related Data
      </label>
      <input type="hidden" name="includeRelated" value="false" />
    </div>
    <button class="btn btn-primary m-1">Filter</button>
  </form>
</div>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th><th>Price</th>
```

```
    @if (ViewBag.includeRelated) {
      <th>Supplier</th>
    }
  </tr>
</thead>
<tbody>
  @foreach (var p in Model) {
    <tr>
      <td>@p.Id</td><td>@p.Name</td>
      <td>@p.Category</td><td>$@p.Price.ToString("F2")</td>
      @if (ViewBag.includeRelated) {
        <td>@p.Supplier?.Name</td>
      }
      <td>
        <form asp-action="Delete" method="post">
          <a asp-action="Edit" class="btn btn-sm btn-warning"
             asp-route-id="@p.Id">
            Edit
          </a>
          <input type="hidden" name="id" value="@p.Id" />
          <button type="submit" class="btn btn-danger btn-sm">
            Delete
          </button>
        </form>
      </td>
    </tr>
  }
</tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create New Product</a>
```

В представление добавлен флажок, который будет управлять тем, должны ли связанные данные включаться в запрос. Если пользователь выбрал включение связанных данных, то свойство `ViewBag.includeRelated` будет равно `true`, и представление применяет это значение для отображения дополнительного столбца в таблице. Значение столбца получается чтением свойства `Name` объекта `Supplier`, связанного с отображаемым объектом `Product`:

```
...
<td>@p.Supplier?.Name</td>
...
```

Представлению Razor ничего не известно об источнике объектов `Supplier`, поскольку Entity Framework Core заполняет навигационные свойства обычными объектами .NET.

Подготовка базы данных

В главе будет показано, как выполнять полный набор операций над связанными данными, но это помогает получить начальные данные для демонстрации работы запросов с методом `Include()`. Чтобы добавить объекты `Supplier` в БД, внесите в класс `SeedData` изменения, приведенные в листинге 14.9.

Листинг 14.9. Заполнение БД начальными данными с объектами Supplier в файле SeedData.cs из папки Models

```
...
private static Product[] Products {
  get {
    Product[] products = new Product[] {
      new Product { Name = "Kayak", Category = "Watersports",
          Price = 275, Color = Colors.Green, InStock = true },
      new Product { Name = "Lifejacket", Category = "Watersports",
          Price = 48.95m, Color = Colors.Red, InStock = true },
      new Product { Name = "Soccer Ball", Category = "Soccer",
          Price = 19.50m, Color = Colors.Blue, InStock = true },
      new Product { Name = "Corner Flags", Category = "Soccer",
          Price = 34.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Stadium", Category = "Soccer",
          Price = 79500, Color = Colors.Red, InStock = true },
      new Product { Name = "Thinking Cap", Category = "Chess",
          Price = 16, Color = Colors.Blue, InStock = true },
      new Product { Name = "Unsteady Chair", Category = "Chess",
          Price = 29.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Human Chess Board", Category = "Chess",
          Price = 75, Color = Colors.Red, InStock = true },
      new Product { Name = "Bling-Bling King", Category = "Chess",
          Price = 1200, Color = Colors.Blue, InStock = true }};

    Supplier s1 = new Supplier { Name = "Surf Dudes",
      City = "San Jose", State = "CA" };
    Supplier s2 = new Supplier { Name = "Chess Kings",
      City = "Seattle", State = "WA" };

    products.First().Supplier = s1;
    foreach (Product p in products.Where(p => p.Category == "Chess")) {
      p.Supplier = s2;
    }
    return products;
  }
}
...
```

Статический массив объектов `Product` в классе `SeedData` заменен свойством, конструкция `get` которого создает два объекта `Supplier` и ассоциирует их с объектами `Product`. Поставщик `Surf Dudes` ассоциируется с первым объектом `Product` (`Kayak`), а поставщик `Chess Kings` — со всеми объектами `Product` в категории `Chess`.

Теперь предстоит подготовка БД для приложения путем применения миграций, включая новую миграцию, которая определяет отношение `Supplier`. Чтобы применить миграции, выполните в папке проекта `DataApp` команду из листинга 14.10.

Листинг 14.10. Подготовка БД

```
dotnet ef database update --context EFDatabaseContext
```

Запустите приложение, открыв окно командной строки, перейдя в папку DataApp и выполнив команду из листинга 14.11.

Листинг 14.11. Запуск примера приложения

```
dotnet run
```

Приложение заполнит БД начальными данными в виде объектов Product и Supplier, что может занять некоторое время. После того, как приложение запустилось, откройте окно браузера и перейдите по ссылке http://localhost:5000. Вы увидите в таблице новый столбец с названиями поставщиков для тех объектов Product, которые связаны с объектом Supplier (рис. 14.3). Обратите внимание, что не со всеми объектами Product связан объект Supplier, а более одного объекта Product имеют отношение с тем же самым объектом Supplier.

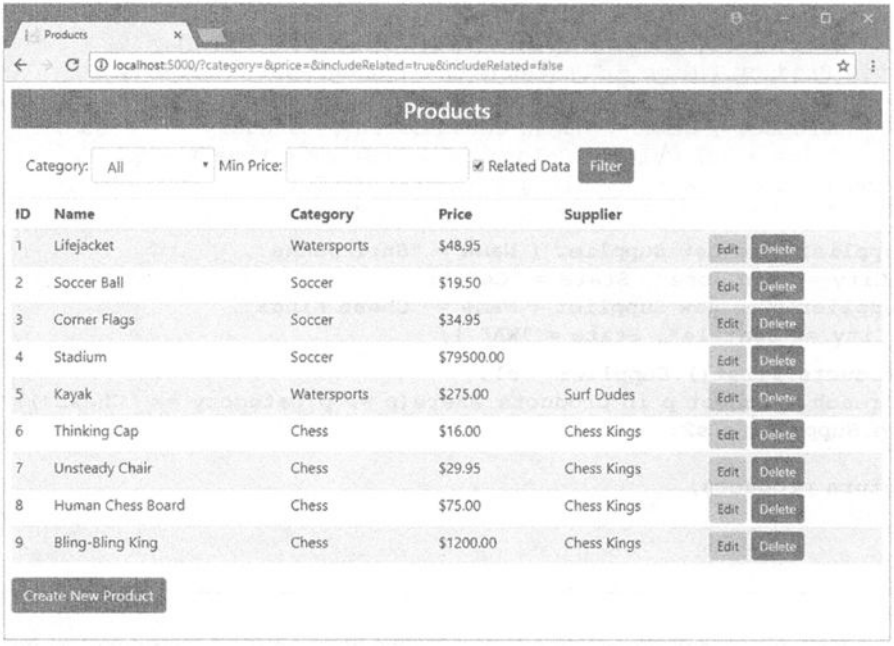

Рис. 14.3. Выполнение примера приложения

Если вы исследуете консольный вывод из приложения, то заметите запрос, который Entity Framework Core использует для получения связанных данных:

```
...
SELECT [p].[Id], [p].[Category], [p].[Color], [p].[InStock], [p].[Name],
      [p].[Price], [p].[SupplierId], [p.Supplier].[Id],
      [p.Supplier].[City], [p.Supplier].[Name], [p.Supplier].[State]
FROM [Products] AS [p]
LEFT JOIN [Supplier] AS [p.Supplier] ON [p].[SupplierId] =
[p.Supplier].[Id]
...
```

Запрос задействует отношение между таблицами Products и Supplier для выполнения соединения, так что все требующиеся данные могут быть получены в единственном запросе. Если снять отметку с флажка Related Data (Связанные данные) и щелкнуть на кнопке Filter (Фильтровать), тогда запрос к БД не будет требовать данные Supplier, а в отображаемой браузером таблице исчезнет столбец Supplier.

Создание и обновление связанных данных

Процесс создания или обновления связанных данных выполняется через навигационные свойства и делается с применением стандартных объектов .NET.

Для примера приложения — подобно большинству реальных проектов — различают три сценария, при которых в БД сохраняется новый объект Supplier или обновляется существующий объект Supplier.

- новые объекты Product и Supplier создаются одновременно;
- создается новый объект Supplier и ассоциируется с существующим объектом Product;
- существующий объект Supplier модифицируется после того, как он был ассоциирован с объектом Product.

Для подготовки к реализации таких операций добавьте в папку Views/Shared файл представления по имени Supplier.cshtml с содержимым, показанным в листинге 14.12.

Листинг 14.12. Содержимое файла Supplier.cshtml из папки Views/Shared

```
@model DataApp.Models.Product

<input type="hidden" asp-for="Supplier.Id" />
<div class="form-group">
  <label asp-for="Supplier.Name"></label>
  <input asp-for="Supplier.Name" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Supplier.City"></label>
  <input asp-for="Supplier.City" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Supplier.State"></label>
  <input asp-for="Supplier.State" class="form-control" />
</div>
```

Моделью для представления является объект Product. В представлении, которое будет использоваться для работы с данными Supplier, это может показаться несколько странным. Однако следование соглашениям MVC и применение вспомогательных функций дескрипторов asp-for настраивает HTML-элементы так, что процесс привязки моделей MVC создаст объект Supplier, используя значения из элементов input, и присвоит его свойству Supplier созданного объекта Product.

Отредактируйте файл представления Editor.cshtml из папки Views/Home, чтобы встроить в него представление, приведенное в листинге 14.12, и придать определенную структуру содержимому представления (листинг 14.13).

Листинг 14.13. Использование частичного представления в файле `Editor.cshtml` из папки `Views/Home`

```
@model DataApp.Models.Product
@{
  ViewData["Title"] = ViewBag.CreateMode ? "Create" : "Edit";
  Layout = "_Layout";
}
<form asp-action="@(ViewBag.CreateMode ? "Create" : "Edit")" method="post">
  <input name="original.Id" value="@Model?.Id" type="hidden" />
  <input name="original.Name" value="@Model?.Name" type="hidden" />
  <input name="original.Category" value="@Model?.Category" type="hidden" />
  <input name="original.Price" value="@Model?.Price" type="hidden" />
  <div class="row m-1">
    <div class="col-6">
      <h5 class="bg-info text-center p-2 text-white">Product</h5>
      <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
      </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>
    <div class="col-6">
      <h5 class="bg-info text-center p-2 text-white">Supplier</h5>
      @Html.Partial("Supplier", Model)
    </div>
  </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>
```

Стили Bootstrap, примененные к элементам `div` в листинге 14.13, создают простую сетку, так что элементы формы отображаются в двух колонках: в одной находятся поля, определенные для класса `Product`, а в другой — поля для класса `Supplier`. Вы увидите итоговую компоновку в следующем разделе.

Создание нового поставщика при создании нового товара

Первый сценарий самый простой: в БД одновременно добавляются новые объекты `Product` и `Supplier`. Никаких дополнительных изменений вносить в приложение не придется, поскольку существующий код, который создает новый объект `Product`, автоматически будет иметь дело с объектами `Supplier`.

Запустите приложение, перейдите в браузере по ссылке `http://localhost:5000` и щелкните на кнопке Create New Product (Создать новый товар).

Заполните элементы формы в обеих колонках, как показано на рис. 14.4, и щелкните на кнопке Save (Сохранить). При желании воссоздать объекты `Product` и `Supplier`, представленные на рис. 14.4, используйте значения из табл. 14.3.

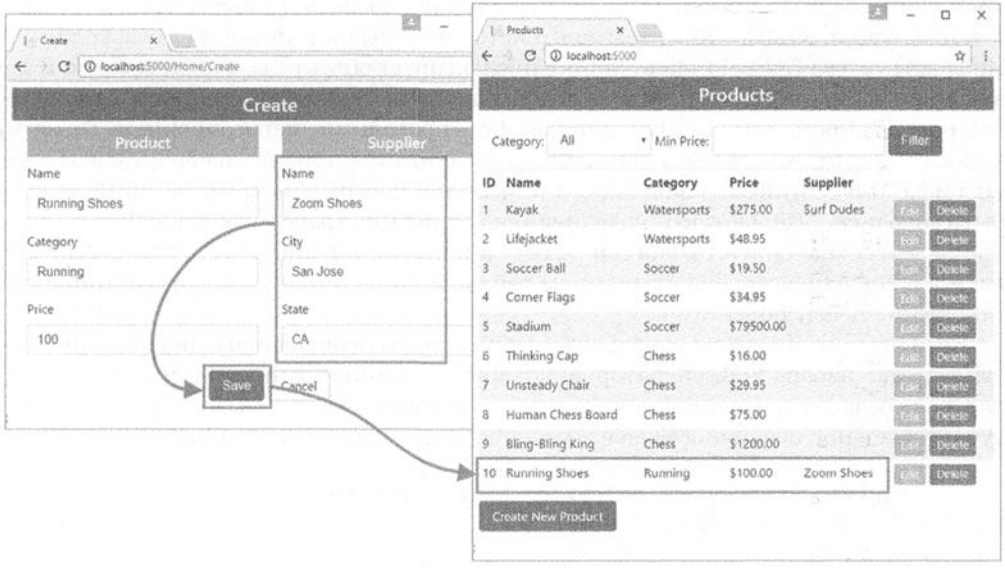

Рис. 14.4. Создание поставщика вместе с новым товаром

Таблица 14.3. Данные о товаре и поставщике

Поле	Значение
Колонка Product (Товар)	
Name (Наименование)	`Running Shoes` (Кроссовки)
Category (Категория)	`Running` (Бег)
Price (Цена)	`100`
Колонка Supplier (Поставщик)	
Name (Название)	`Zoom Shoes`
City (Город)	`San Jose` (Сан-Хосе)
State (Штат)	`CA` (Калифорния)

После щелчка на кнопке Save браузер отправит серверу HTTP-запрос `POST`, который содержит значения из элементов `input`, определенных в листинге 14.12. Такие элементы `input` визуализировались механизмом Razor, поэтому они будут распознаваться связывателем моделей MVC как свойства для объекта `Supplier`. Например, вот как визуализируется элемент для свойства `City`:

```
...
<input class="form-control" type="text"
       id="Supplier_City" name="Supplier.City" value="">
...
```

Связыватель моделей применяет значения в HTML-форме для создания объекта `Product` и объекта `Supplier`, который присваивается навигационному свойству `Supplier` объекта `Product`. Объект `Product` используется в качестве аргумента метода `Create()` контроллера `Home`, который передается методу `CreateProduct()` хранилища. Метод `CreateProduct()` хранилища добавляет объект `Product` в коллекцию, управляемую объектом контекста, и вызывает метод `SaveChanges()`. Инфраструктура Entity Framework Core инспектирует объект `Product` и выясняет, что значение `Id` равно нулю; это служит признаком нового объекта, подлежащего сохранению в БД. Кроме того, Entity Framework Core проходит по навигационному свойству `Product.Supplier` с целью инспектирования объекта `Supplier` и обнаруживает, что он также имеет нулевое значение `Id` и является новым объектом, который должен быть сохранен в БД. Объекты сохраняются как новые строки в таблицах `Products` и `Supplier`, а значение столбца `SupplierId` в строке таблицы `Products` устанавливается в значение столбца `Id` в строке таблицы `Supplier`, чтобы зарегистрировать отношение между объектами.

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то увидите два набора SQL-операторов, которые сохранили данные. Первая пара операторов создает строку в таблице `Supplier`, которая хранит объект `Supplier`, и получает значение первичного ключа, назначенное сервером баз данных:

```
...
INSERT INTO [Supplier] ([City], [Name], [State])
VALUES (@p0, @p1, @p2);

SELECT [Id]
FROM [Supplier]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

Вторая пара операторов создает строку в таблице `Products` и получает значение первичного ключа, назначенное сервером баз данных:

```
...
INSERT INTO [Products] ([Category], [Color], [InStock],
  [Name], [Price], [SupplierId])
VALUES (@p3, @p4, @p5, @p6, @p7, @p8);

SELECT [Id]
FROM [Products]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

В результате инфраструктура Entity Framework Core гладко работает с объектами, которые были созданы связывателем моделей MVC, и сохраняет в БД две новые строки для представления новых объектом вместе с отношением между ними (рис. 14.5).

Обновление поставщика при обновлении товара

Для создания или обновления объекта `Supplier`, когда объект `Product`, с которым он связан, уже был сохранен в БД, потребуется внести изменение в класс контекста. Причина связана с приемом, который применялся ранее, чтобы задействовать в своих интересах средство обнаружения изменений. Это средство инфраструктура Entity Framework Core предлагает для сокращения количества запросов к БД, которые выполняются и должны быть расширены с целью включения данных `Supplier`. Для включения данных `Supplier` в HTML-форму, отправляемую клиенту, добавьте в представление `Supplier.cshtml` элементы, выделенные полужирным в листинге 14.14.

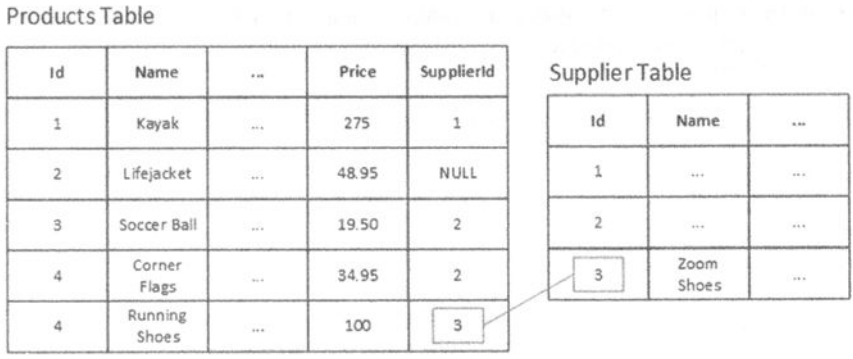

Рис. 14.5. Создание новых объектов `Product` и `Supplier`

Листинг 14.14. Добавление существующих значений в файле `Supplier.cshtml` из папки `Views/Shared`

```
@model DataApp.Models.Product

<input name="original.Supplier.Id" value="@Model.Supplier?.Id"
type="hidden" />
<input name="original.Supplier.Name" value="@Model.Supplier?.Name"
type="hidden" />
<input name="original.Supplier.City" value="@Model.Supplier?.City"
type="hidden" />
<input name="original.Supplier.State" value="@Model.Supplier?.State"
type="hidden" />

<input type="hidden" asp-for="Supplier.Id" />
<div class="form-group">
  <label asp-for="Supplier.Name"></label>
  <input asp-for="Supplier.Name" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Supplier.City"></label>
  <input asp-for="Supplier.City" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Supplier.State"></label>
  <input asp-for="Supplier.State" class="form-control" />
</div>
```

Скрытые элементы `input` являются аналогами таких элементов в представлении `Editor.cshtml`. Чтобы обеспечить сохранение к БД внесенных пользователем изменений, добавьте в метод `UpdateProduct()` класса `EFDataRepository` операторы, выделенные полужирным в листинге 14.15.

Добавленные операторы копируют значения `Name`, `City` и `State` в объект, который отслеживается инфраструктурой Entity Framework Core, тем самым гарантируя сохранение измененных значений в БД. Обратите внимание, что свойство `Supplier.Id` не копировалось; его значение управляется Entity Framework Core или сервером баз данных и при выполнении обновления его лучше не затрагивать.

Листинг 14.15. Включение данных о поставщике в файле `EFDataRepository.cs` из папки `Models`

```
...
public void UpdateProduct(Product changedProduct, Product
originalProduct = null) {
  if (originalProduct == null) {
    originalProduct = context.Products.Find(changedProduct.Id);
  } else {
    context.Products.Attach(originalProduct);
  }
  originalProduct.Name = changedProduct.Name;
  originalProduct.Category = changedProduct.Category;
  originalProduct.Price = changedProduct.Price;
  originalProduct.Supplier.Name = changedProduct.Supplier.Name;
  originalProduct.Supplier.City = changedProduct.Supplier.City;
  originalProduct.Supplier.State = changedProduct.Supplier.State;
  context.SaveChanges();
}
...
```

Чтобы протестировать изменения, перезапустите приложение, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit (Редактировать) для товара `Thinking Cap`. Внесите изменения согласно табл. 14.4 для просмотра результатов обновления связанных объектов.

Таблица 14.4. Значения для редактирования сведений о товаре `Thinking Cap` и его поставщике

Поле	Значение
Name (Имя) в колонке Product (Товар)	`Thinking Cap (Medium)`
Name (Имя) в колонке Supplier (Поставщик)	`The Pawn Brokers`
City (Город)	`Chicago` (Чикаго)
State (Штат)	`IL` (Иллинойс)

Щелкните на кнопке Save; браузер отправит модифицированные данные приложению. Связыватель моделей создаст объекты `Product` и `Supplier` из данных запроса, а Entity Framework Core сохранит изменения в БД. Поскольку объект `Supplier` связан со всеми объектами `Product` в категории `Chess`, изменения отразятся в нескольких строках таблицы (рис. 14.6).

Удаление связанных данных

По умолчанию инфраструктура Entity Framework Core не будет следовать по навигационному свойству для удаления связанных данных из БД. Это означает, что удаление объекта `Product` не приводит к удалению связанного объекта `Supplier`.

Причина такого поведения в том, что с объектом `Supplier` могут быть связаны другие объекты `Product`, и его удаление из БД внесло бы несогласованность в данные, чего сервер баз данных всячески стремится избежать.

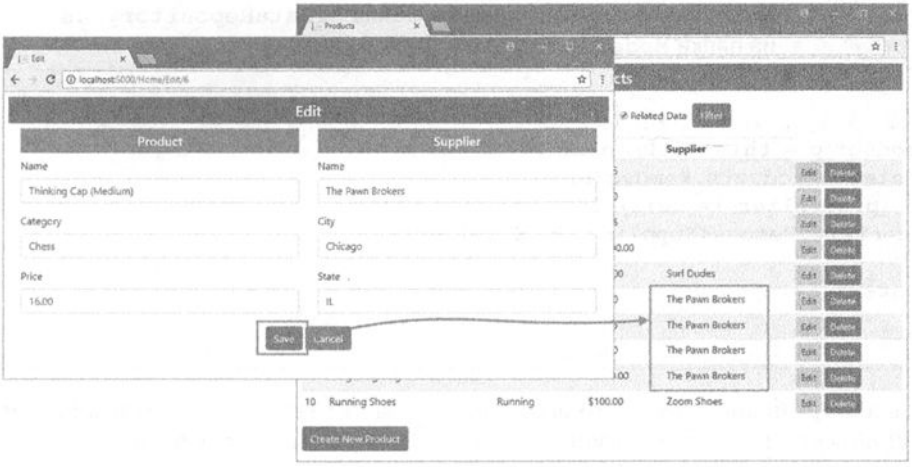

Рис. 14.6. Обновление товара и поставщика

Доступны средства, упрощающие выполнение операций удаления; они будут описаны в последующих главах, а пока цель заключается в демонстрации стандартной конфигурации и объяснения скрывающейся в ней ловушки.

Чтобы увидеть, как обрабатывается операция удаления, запустите приложение, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete (Удалить) для товара `Unsteady Chair`, который связан с поставщиком `The Pawn Brokers`. Операция удалит объект `Product`, но оставит объект `Supplier`, с которым по-прежнему связаны другие объекты `Product` в категории `Chess` (рис. 14.7).

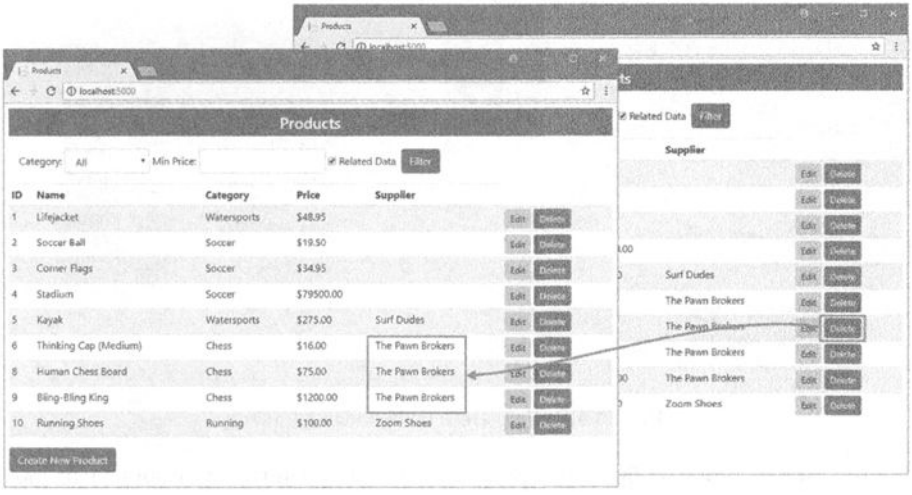

Рис. 14.7. Удаление объекта, имеющего связанные данные

Вы можете указать инфраструктуре Entity Framework Core, что нужно удалить связанные данные, но сервер баз данных сгенерирует исключение, если такое удаление привело бы к несогласованности БД. Чтобы посмотреть, как все работает, внесите в метод `DeleteProduct()` класса `EFDataRepository` изменения, показанные в листинге 14.16.

Листинг 14.16. Удаление связанных данных в файле `EFDataRepository.cs` из папки `Models`

```
...
public void DeleteProduct(long id) {
  Product p = this.GetProduct(id);
  context.Products.Remove(p);
  if (p.Supplier != null) {
    context.Remove<Supplier>(p.Supplier);
  }
  context.SaveChanges();
}
...
```

Операторы в листинге 14.16 вызывают метод `GetProduct()`, который извлекает из БД объект `Product` и связанный с ним объект `Supplier`. Объект `Product` удаляется посредством метода `Remove()`, обеспечиваемого свойством `DbSet<Product>` объекта контекста. Прямой доступ к данным `Supplier` отсутствует, но для удаления любого объекта из БД можно использовать метод `Remove<T>()` объекта контекста:

```
...
context.Remove<Supplier>(p.Supplier);
...
```

Чтобы протестировать код, перезапустите приложение, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete для товара `Running Shoes` (рис. 14.8). Учитывая, что это единственный объект `Product`, с которым связан объект `Supplier` с названием `Zoom Shoes`, удаление обоих объектов не приводит к несогласованности, а потому они благополучно удалятся.

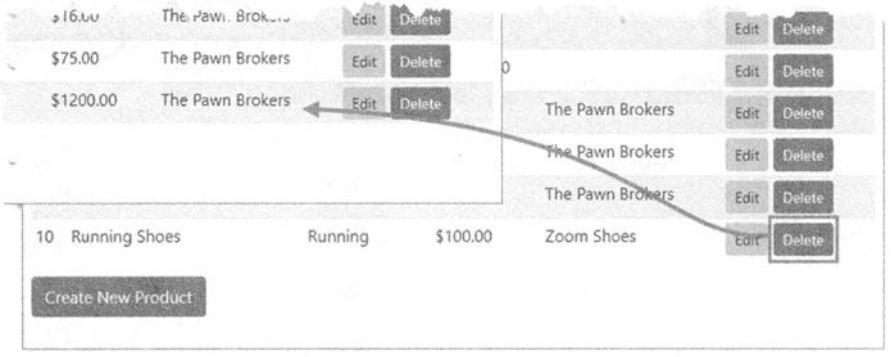

Рис. 14.8. Удаление связанных данных

Для просмотра, каким образом сервер баз данных препятствует образованию несогласованности, щелкните на кнопке Delete для товара `Human Chess Board`. Поскольку объект `Supplier`, связанный с данным объектом `Product`, также связан с другими объектами `Product`, его удаление оставило бы строки таблицы `Products` со значениями в столбце `SupplierId`, которое ссылается на несуществующую строку в таблице `Supplier`. На рис. 14.9 видно, что сервер баз данных сгенерировал исключение и не выполнил операцию, которая вызвала бы описанную проблему.

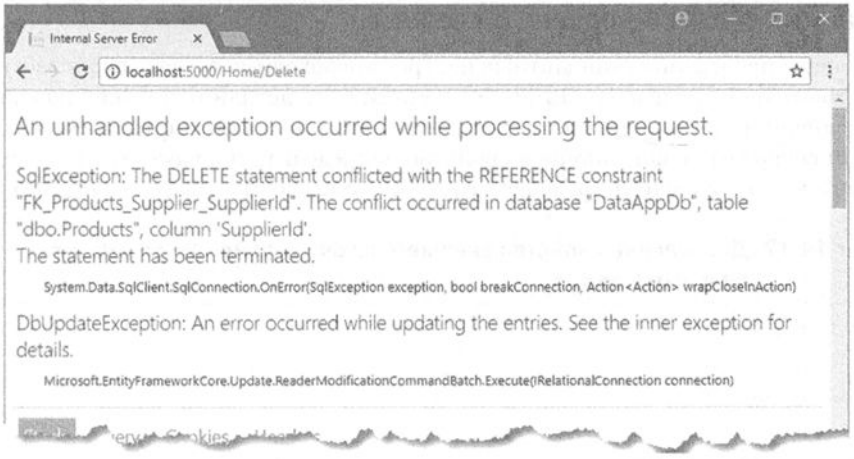

Рис. 14.9. Операция удаления, которая привела бы к несогласованности БД

В настоящее время операция удаления не особенно полезна, но позже в главе будет показано, как применять другой тип отношения и делать операции удаления более предсказуемыми.

Создание обязательного отношения

Классы `Product` и `Supplier` связаны необязательным отношением, которое говорит о том, что объект `Product` *не обязан* быть связанным с объектом `Supplier`. Такой тип отношения является стандартным и отражает особенности работы объектов в части MVC приложения, когда свойство, ссылающееся на другой объект, может быть `null`.

Определенные отношения должны быть более формальными, где важно, чтобы объект одного типа всегда был связан с объектом какого-то другого типа. В таких ситуациях можно создавать *обязательное отношение*, которое реконфигурирует БД для навязывания отношения от имени приложения.

Создание обязательного отношения означает сообщение инфраструктуре Entity Framework Core о том, что она должна создать столбец внешнего ключа, который отслеживает отношение в БД. Ниже представлены операторы из миграции, создающие отношение между таблицами `Products` и `Supplier`:

```
...
migrationBuilder.AddColumn<long>(name: "SupplierId",
  table: "Products", nullable: true);
...
migrationBuilder.AddForeignKey(name: "FK_Products_Supplier_SupplierId",
  table: "Products", column: "SupplierId", principalTable: "Supplier",
  principalColumn: "Id", onDelete: ReferentialAction.Restrict);
...
```

Инфраструктура Entity Framework Core не была снабжена какой-либо информацией об отношении внешнего ключа и использовала разумные стандартные настройки, в том числе разрешение значений `null` в столбце, отслеживающем отношение, и создание отношения внешнего ключа с именем, которое объединяет имена связанных таблиц и столбцов.

Создание свойства внешнего ключа

Для переопределения стандартных настроек необходимо создать свойство, которое предоставит инфраструктуре Entity Framework Core детали о том, как должен быть создан внешний ключ. Речь идет о *свойстве внешнего ключа*, определяемом в классе, который содержит навигационное свойство. Определите в классе `Product` свойство внешнего ключа, конфигурирующее отношение с классом `Supplier` (листинг 14.17).

Листинг 14.17. Добавление свойства внешнего ключа в файле `Product.cs` из папки `Models`

```
namespace DataApp.Models {

  public enum Colors {
    Red, Green, Blue
  }

  public class Product {

    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    public Colors Color { get; set; }
    public bool InStock { get; set; }

    public long SupplierId { get; set; }
    public Supplier Supplier { get; set; }
  }
}
```

Имя свойства сообщает инфраструктуре Entity Framework Core о том, к какому навигационному свойству оно относится, за счет комбинирования имени навигационного свойства либо имени связанного класса с именем первичного ключа. В случае класса `Product` оба соглашения приводят к свойству под названием `SupplierId`, которое Entity Framework Core будет опознавать как свойство внешнего ключа для навигационного свойства `Supplier`. (В главе 18 будет показано, как переопределить соглашение, приятое для имен свойств.)

Тип свойства внешнего ключа указывает инфраструктуре Entity Framework Core на обязательность отношения. Отношение будет необязательным, если свойство внешнего ключа может быть установлено в `null` (когда выбран тип наподобие `long?`), и обязательным, если значения `null` невозможны (когда выбран тип вроде `long`). В листинге 14.17 для свойства был применен тип `long`, что заставляет инфраструктуру Entity Framework Core создать обязательное отношение, т.к. свойство `long` устанавливать в `null` нельзя.

Выполните в папке проекта `DataApp` команду из листинга 14.18 для создания миграции, которая модифицирует БД с целью применения атрибута.

Листинг 14.18. Создание миграции

```
dotnet ef migrations add Required --context EFDatabaseContext
```

Если вы просмотрите операторы в методе `Up()` внутри файла `<отметка времени>_Required.cs` из папки `Migrations`, то увидите, каким образом изменение будет применяться к БД:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.DropForeignKey(name: "FK_Products_Supplier_SupplierId",
    table: "Products");

  migrationBuilder.AlterColumn<long>(name: "SupplierId", table: "Products",
    nullable: false, oldClrType: typeof(long), oldNullable: true);

  migrationBuilder.AddForeignKey(name: "FK_Products_Supplier_SupplierId",
    table: "Products", column: "SupplierId", principalTable: "Supplier",
    principalColumn: "Id", onDelete: ReferentialAction.Cascade);
}
...
```

Миграция заново создаст отношение внешнего ключа между таблицами `Products` и `Supplier` для отражения нового свойства в классе `Product`. Самым важным оператором является тот, который изменяет столбец `SupplierId`, и его аргумент `nullable` запрещает значения `null`:

```
...
migrationBuilder.AlterColumn<long>(name: "SupplierId", table: "Products",
  nullable: false, oldClrType: typeof(long), oldNullable: true);
...
```

В измененном столбце больше не разрешается хранить значения `null` и внешний ключ требует, чтобы значения соответствовали строкам в таблице `Supplier`; именно так управляются обязательные отношения.

Удаление базы данных и подготовка начальных данных

Если вы попытаетесь применить миграцию к БД, то получите ошибку, потому что в таблице `Products` присутствуют строки, которые содержат значения `null` и больше не являются допустимыми.

В реальном проекте вы должны подготовить БД к миграции, удалив все данные, которые не удовлетворяют требованиям. В рассматриваемом примере приложения проще полностью удалить БД, обновить начальные данные и затем воссоздать БД, обеспечив допустимость всех данных. Первым делом выполните в папке проекта `DataApp` команду из листинга 14.19, чтобы удалить БД.

Листинг 14.19. Удаление БД

```
dotnet ef database drop --force --context EFDatabaseContext
```

Далее обновите конструкцию `get` для свойства `Products`, определенного в классе `SeedData`, чтобы все сохраненные объекты `Product` были связаны с объектами `Supplier` (листинг 14.20).

Листинг 14.20. Определение отношения между данными в файле `SeedData.cs` из папки `Models`

```
...
private static Product[] Products {
```

```
  get {
    Product[] products = new Product[] {
      new Product { Name = "Kayak", Category = "Watersports",
          Price = 275, Color = Colors.Green, InStock = true },
      new Product { Name = "Lifejacket", Category = "Watersports",
          Price = 48.95m, Color = Colors.Red, InStock = true },
      new Product { Name = "Soccer Ball", Category = "Soccer",
          Price = 19.50m, Color = Colors.Blue, InStock = true },
      new Product { Name = "Corner Flags", Category = "Soccer",
          Price = 34.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Stadium", Category = "Soccer",
          Price = 79500, Color = Colors.Red, InStock = true },
      new Product { Name = "Thinking Cap", Category = "Chess",
          Price = 16, Color = Colors.Blue, InStock = true },
      new Product { Name = "Unsteady Chair", Category = "Chess",
          Price = 29.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Human Chess Board", Category = "Chess",
          Price = 75, Color = Colors.Red, InStock = true },
      new Product { Name = "Bling-Bling King", Category = "Chess",
          Price = 1200, Color = Colors.Blue, InStock = true });
      Supplier acme = new Supplier { Name = "Acme Co",
        City = "New York", State = "NY" };
      Supplier s1 = new Supplier { Name = "Surf Dudes",
        City = "San Jose", State = "CA" };
      Supplier s2 = new Supplier { Name = "Chess Kings",
        City = "Seattle", State = "WA" };
      foreach (Product p in products) {
        if (p == products[0]) {
          p.Supplier = s1;
        } else if (p.Category == "Chess") {
          p.Supplier = s2;
        } else {
          p.Supplier = acme;
        }
      }
      return products;
    }
}
...
```

Изменения в начальных данных поддерживают ранее настроенное отношение и используют поставщика `Acme Co` для всех остальных товаров.

Обновление и заполнение начальными данными базы данных

Выполните в папке проекта `DataApp` команду из листинга 14.21 для подготовки БД, включая обеспечение обязательного отношения между объектами `Product` и `Supplier`.

Листинг 14.21. Обновление БД для включения измененного отношения

```
dotnet ef database update --context EFDatabaseContext
```

После того как инфраструктура Entity Framework Core применит миграции, запустите приложение, используя `dotnet run`, и перейдите в браузере по ссылке `http://localhost:5000`. Во время запуска приложения БД заполнится начальными данными, которые удовлетворяют ограничениям, примененным к БД, так что все объекты `Product` будут связаны с объектами `Supplier` (рис. 14.10).

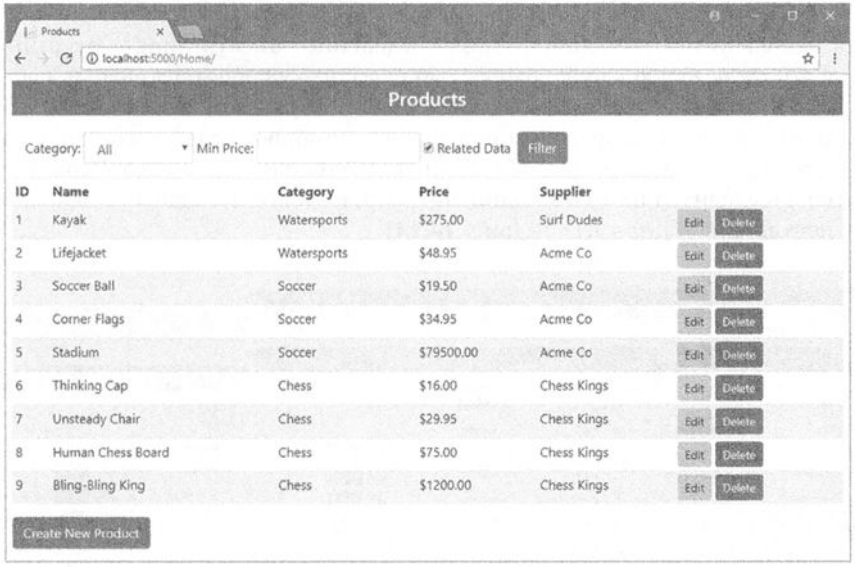

Рис. 14.10. Создание отношения

Операция удаления при наличии обязательного отношения

Свойство внешнего ключа приводит к двум важным изменениям в миграции, созданной в листинге 14.21. Первое изменение касается запрета хранения значений `null` в столбце `SupplierId`, что превращает необязательное отношение в обязательное. Второе изменение порождает неожиданное отличие в способе работы БД и вызывает много путаницы.

Когда в листинге 14.3 в класс `Product` было добавлено навигационное свойство до определения свойства внешнего ключа, созданная миграция конфигурировала отношение внешнего ключа следующим образом:

```
...
migrationBuilder.AddForeignKey(
  name: "FK_Products_Supplier_SupplierId", table: "Products",
  column: "SupplierId", principalTable: "Supplier",
  principalColumn: "Id", onDelete: ReferentialAction.Restrict);
...
```

Аргумент `onDelete` с использованием перечисления `ReferentialAction` сообщает БД, что делать при удалении строки в таблице `Supplier`, от которой зависит строка таблицы `Products`. Для необязательных отношений применяется значение `Restrict`, конфигурирующее БД так, что строка `Supplier` не может быть удалена, если в таблице `Products` имеются строки, которые от нее зависят. По этой причине при попытке удаления строки `Supplier` ранее в главе возникала ошибка.

В случае создания обязательного отношения внешний ключ реконфигурируется с другим значением `ReferentialAction`:

```
...
migrationBuilder.AddForeignKey(
  name: "FK_Products_Supplier_SupplierId", table: "Products",
  column: "SupplierId", principalTable: "Supplier",
  principalColumn: "Id", onDelete: ReferentialAction.Cascade);
...
```

Когда используется значение `Cascade`, удаление объекта `Supplier` становится причиной *каскадного удаления*, которое означает, что удаляются также и любые другие объекты `Product`, зависящие от этого объекта `Supplier`. Чтобы увидеть эффект от каскадного удаления, запустите приложение с помощью `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete для товара `Unsteady Chair`. Теперь удалится не только данный товар, но и все остальные товары, относящиеся к поставщику `Chess Kings` (рис. 14.11).

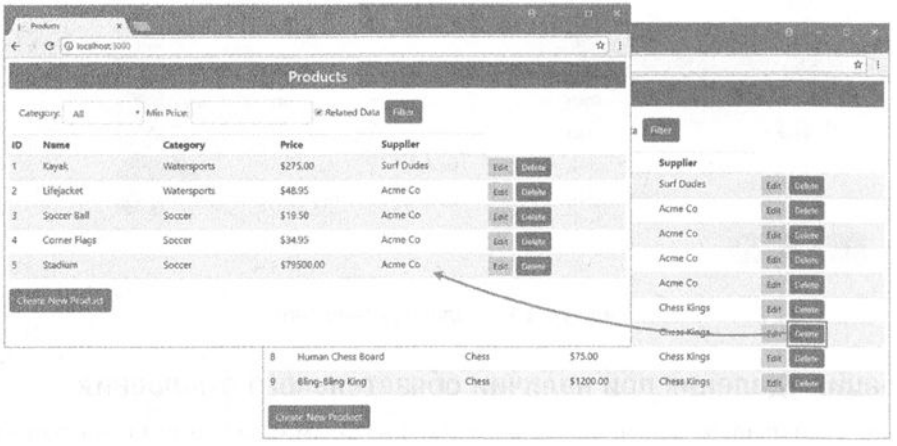

Рис. 14.11. Эффект от каскадного удаления

Важно понимать, что именно действие по удалению объекта `Supplier` инициирует каскадное удаление. Можно удалить любой объект `Product` из БД, но объекты `Supplier` останутся незатронутыми. При удалении объекта `Supplier` каскадное удаление несет ответственность за удаление из таблицы `Products` строк, которые имеют отношение внешнего ключа со строкой, удаляемой из таблицы `Supplier`.

Имейте в виду, что каскадное удаление выполняет сервер баз данных, а не инфраструктура Entity Framework Core. Конфигурация БД, которая была применена миграцией, сообщает серверу баз данных, что делать, когда строка удаляется. В главе 22 будет объясняться, как взять под свой контроль такое поведение.

Выполнение запросов для множества отношений

Большинство приложений содержат более одного отношения между данными. Для создания, обновления и удаления более сложных связанных данных используется тот же самый процесс, но нужно применять другой прием сообщения инфраструктуре Entity Framework Core о следовании по всем навигационным свойствам, чтобы обеспечить получение всех требующихся данных.

В целях демонстрации добавьте в папку Models файл класса по имени ContactLocation.cs с кодом, показанным в листинге 14.22.

Листинг 14.22. Содержимое файла ContactLocation.cs из папки Models

```
namespace DataApp.Models {
  public class ContactLocation {
    public long Id { get; set; }
    public string LocationName { get; set; }
    public string Address { get; set; }
  }
}
```

Затем добавьте в папку Models файл класса по имени ContactDetails.cs и поместите в него код из листинга 14.23.

Листинг 14.23. Содержимое файла ContactDetails.cs из папки Models

```
namespace DataApp.Models {
  public class ContactDetails {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Phone { get; set; }
    public ContactLocation Location { get; set; }
  }
}
```

В классе ContactDetails определено навигационное свойство Location, которое создает отношение с объектом ContactLocation. В заключение добавьте в класс Supplier навигационное свойство, как показано в листинге 14.24.

Листинг 14.24. Добавление навигационного свойства в файле Supplier.cs из папки Models

```
namespace DataApp.Models {
  public class Supplier {
    public long Id { get; set; }
    public string Name { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public ContactDetails Contact { get; set; }
  }
}
```

Свойство Contact создает отношение с объектом ContactDetails. Общим результатом будет цепочка отношений между классами (рис. 14.12).

Обновление и заполнение начальными данными базы данных

Чтобы создать миграцию, которая позволит БД хранить экземпляры новых классов, выполните команду из листинга 14.25, находясь в папке проекта DataApp.

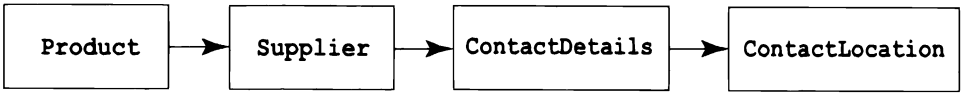

Рис. 14.12. Создание цепочки навигационных свойств

Совет. Если вы нарушили последовательность команд, приведенных в настоящей главе, и начинаете получать исключения "Invalid Column Name" ("Недопустимое имя столбца"), тогда закомментируйте операторы в классе `Startup`, заполняющие БД начальными данными, и выполните команды из листингов 14.26 и 14.25. После того, как БД обновится, удалите комментарии с операторов в классе `Startup` и запустите приложение.

Листинг 14.25. Создание миграции

```
dotnet ef migrations add AdditionalTypes --context EFDatabaseContext
```

Просмотрев операторы C# в файле `<отметка времени>_AdditionalTypes.cs`, который был создан в папке `Migrations`, вы заметите, что инфраструктура Entity Framework Core последовала соглашениям, обсужденным ранее в главе, для создания таблиц, предназначенных для хранения объектов `ContentDetails` и `ContactLocation`.

Чтобы удалить и воссоздать БД, сделав возможным повторное заполнение начальными данными, выполните в папке проекта `DataApp` команды из листинга 14.26.

Листинг 14.26. Воссоздание БД

```
dotnet ef database drop --force --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Расширьте начальные данные, которыми должна заполняться БД, добавив операторы из листинга 14.27 в конструкцию `get` свойства `Products` в классе `SeedData`.

Листинг 14.27. Расширение начальных данных в файле `SeedData.cs` из папки `Models`

```
...
private static Product[] Products {
  get {
    Product[] products = new Product[] {
      new Product { Name = "Kayak", Category = "Watersports",
          Price = 275, Color = Colors.Green, InStock = true },
      new Product { Name = "Lifejacket", Category = "Watersports",
          Price = 48.95m, Color = Colors.Red, InStock = true },
      new Product { Name = "Soccer Ball", Category = "Soccer",
          Price = 19.50m, Color = Colors.Blue, InStock = true },
      new Product { Name = "Corner Flags", Category = "Soccer",
          Price = 34.95m, Color = Colors.Green, InStock = true },
      new Product { Name = "Stadium", Category = "Soccer",
          Price = 79500, Color = Colors.Red, InStock = true },
      new Product { Name = "Thinking Cap", Category = "Chess",
          Price = 16, Color = Colors.Blue, InStock = true },
      new Product { Name = "Unsteady Chair", Category = "Chess",
          Price = 29.95m, Color = Colors.Green, InStock = true },
```

```
      new Product { Name = "Human Chess Board", Category = "Chess",
          Price = 75, Color = Colors.Red, InStock = true },
      new Product { Name = "Bling-Bling King", Category = "Chess",
          Price = 1200, Color = Colors.Blue, InStock = true });
    ContactLocation hq = new ContactLocation {
      LocationName = "Corporate HQ", Address = "200 Acme Way"
    };
    ContactDetails bob = new ContactDetails {
      Name = "Bob Smith", Phone = "555-107-1234", Location = hq
    };
    Supplier acme = new Supplier { Name = "Acme Co",
      City = "New York", State = "NY", Contact = bob };
    Supplier s1 = new Supplier { Name = "Surf Dudes",
      City = "San Jose", State = "CA" };
    Supplier s2 = new Supplier { Name = "Chess Kings",
      City = "Seattle", State = "WA" };
    foreach (Product p in products) {
      if (p == products[0]) {
        p.Supplier = s1;
      } else if (p.Category == "Chess") {
        p.Supplier = s2;
      } else {
        p.Supplier = acme;
      }
    }
    return products;
  }
}
...
```

Внесенные изменения обеспечивают создание нового контакта для `Bob Smith` в головных офисах и назначают его в качестве контакта поставщику `Acme Co`.

Выполнение запросов к цепочке навигационных свойств

Метод `ThenInclude()` используется для расширения области действия запроса, чтобы следовать по навигационным свойствам, которые определены типами, выбранными с применением метода `Include()`. В листинге 14.28 с помощью метода `ThenInclude()` инфраструктуре Entity Framework Core сообщается о том, что она должна проследовать по навигационным свойствам, определенным классами `Supplier` и `ContactDetails`, чтобы в запрос были включены все отношения, показанные на рис. 14.12.

Листинг 14.28. Следование по навигационным свойствам в файле `EFDatabaseRepository.cs` из папки `Models`

```
...
public Product GetProduct(long id) {
  return context.Products.Include(p => p.Supplier)
    .ThenInclude(s => s.Contact).ThenInclude(c => c.Location)
    .First(p => p.Id == id);
}
...
```

Аргументом метода `ThenInclude()` является лямбда-функция, которая оперирует на типе, выбранном предыдущим вызовом метода `Include()` или `ThenInclude()`, и выбирает навигационное свойство для следования по нему. За счет объединения вызовов `Include()` и `ThenInclude()` можно проходить по сложной модели данных, включая в состав запроса все требующиеся данные.

Чтобы отобразить дополнительные связанные данные, добавьте в представление `Supplier.cshtml` элементы из листинга 14.29.

Листинг 14.29. Отображение связанных данных в файле `Supplier.cshtml` из папки `Views/Shared`

```
@model DataApp.Models.Product

<input name="original.Supplier.Id" value="@Model.Supplier?.Id"
type="hidden" />
<input name="original.Supplier.Name" value="@Model.Supplier?.Name"
type="hidden" />
<input name="original.Supplier.City" value="@Model.Supplier?.City"
type="hidden" />
<input name="original.Supplier.State" value="@Model.Supplier?.State"
type="hidden" />

<input type="hidden" asp-for="Supplier.Id" />
<div class="form-group">
  <label asp-for="Supplier.Name"></label>
  <input asp-for="Supplier.Name" class="form-control" />
</div>

<div class="form-group">
  <label asp-for="Supplier.City"></label>
  <input asp-for="Supplier.City" class="form-control" />
</div>

<div class="form-group">
  <label asp-for="Supplier.State"></label>
  <input asp-for="Supplier.State" class="form-control" />
</div>

@if (Model.Supplier?.Contact != null) {
  <div class="form-group">
    <label asp-for="Supplier.Contact.Name"></label>
    <input asp-for="Supplier.Contact.Name" class="form-control" readonly />
  </div>

  <div class="form-group">
    <label asp-for="Supplier.Contact.Phone"></label>
    <input asp-for="Supplier.Contact.Phone" class="form-control" readonly />
  </div>

  <div class="form-group">
    <label asp-for="Supplier.Contact.Location.LocationName"></label>
    <input asp-for="Supplier.Contact.Location.LocationName"
           class="form-control" readonly />
  </div>
}
```

В представление были добавлены элементы только для чтения, которые отображают значения из расширенных связанных данных. Чтобы протестировать запрос, запустите приложение посредством `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit для одного из товаров от поставщика `Acme Co`. Вы увидите детали контакта (рис. 14.13).

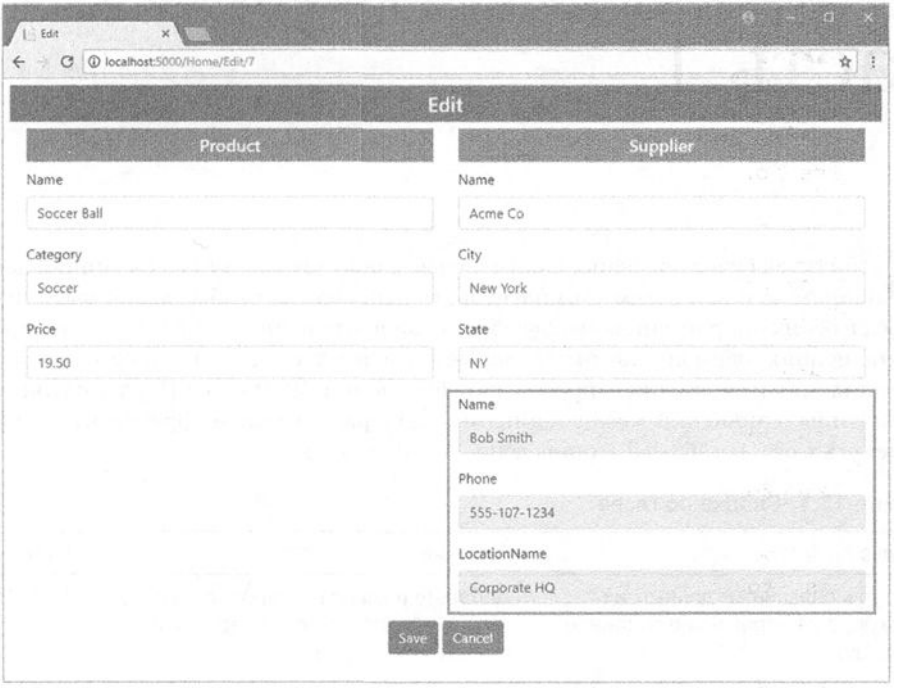

Рис. 14.13. Следование по цепочке навигационных свойств в запросе

Резюме

В главе было показано, как использовать навигационные свойства для создания отношений между данными, а также каким образом они отражаются в БД и потребляются инфраструктурой Entity Framework Core. В следующей главе речь пойдет о применении различных возможностей отношений.

ГЛАВА 15

Работа с отношениями, часть 1

В этой главе будет показано, каким образом получать доступ к связанным данным напрямую и как затем укомплектовать отношение, чтобы можно было перемещаться в обоих направлениях между объектами в отношении. Глава сконцентрирована на отношениях "один ко многим", которые легче всего создавать, но которые требуют внимания при выполнении запросов или обновлении БД. В главе 16 рассматриваются другие типы отношений между данными, поддерживаемые инфраструктурой Entity Framework Core. В табл. 15.1 приведена сводка по главе.

Таблица 15.1. Сводка по главе

Задача	Решение	Листинг
Доступ к связанным данным напрямую, а не через навигационное свойство	Добавьте в класс контекста свойство `DbSet<T>`, после чего создайте и примените миграцию	15.1–15.7
Доступ к связанным данным по типу	Используйте метод `DbContext.Set<T>()`	15.8–15.12
Навигация по отношению в обоих направлениях	Укомплектуйте отношение с помощью навигационного свойства	15.13–15.15, 15.20–15.33
Принудительное выполнение запроса	Применяйте явную загрузку	15.16
Формирование отношений между данными через множество запросов	Положитесь на средство исправления	15.17–15.19

Подготовительные шаги

В главе продолжается работа с проектом `DataApp`, созданным в главе 11 и модифицированным в последующих главах. В качестве подготовки откройте окно командной строки, перейдите в папку `DataApp` и выполните команды из листинга 15.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 15.1. Переустановка БД

```
dotnet ef database drop --force --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Команды удаляют и воссоздают БД, используемую для хранения объектов `Product`, что поможет обеспечить получение ожидаемых результатов в примерах, приводимых в главе. Запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`. Приложение заполнит БД начальными данными и отобразит список товаров (рис. 15.1).

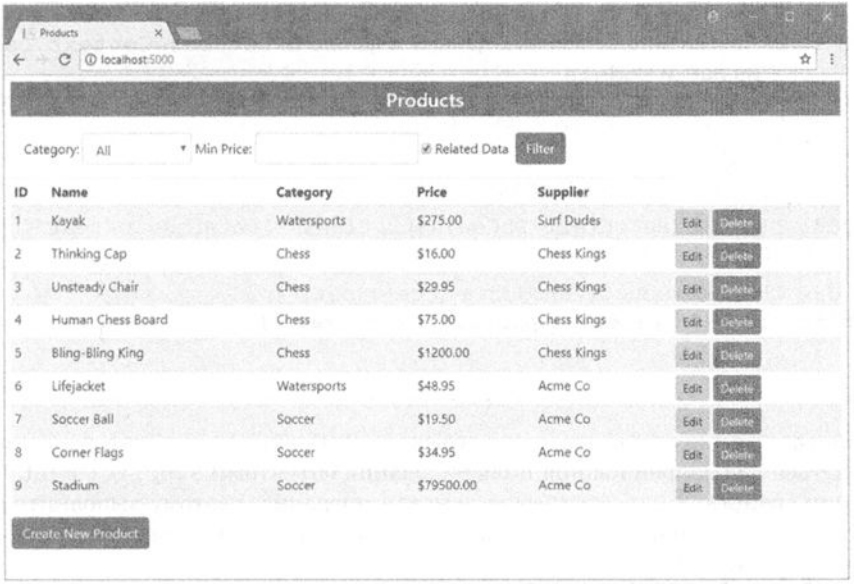

Рис. 15.1. Выполнение примера приложения

Доступ к связанным данным напрямую

В большинстве приложений, по крайней мере, определенные связанные данные имеют собственный жизненный цикл и рабочие потоки, которые пользователи должны выполнять. Например, в приложении `DataApp` легко представить, что администратору может понадобиться внести изменение, отражающее смену адреса. Сделать это сложно, поскольку получать доступ к данным в БД можно, только начиная с объекта `Product` и следуя по его навигационным свойствам. Скажем, если нужно обновить объект `ContactLocation`, то придется начать с поиска объекта `Product`, который связан с объектом `Supplier`, подлежащим модификации, запросить найденный объект `Product` и проследовать по навигационным свойствам для получения объектов `Supplier` и `ContactDetails`. Лишь тогда появится возможность доступа к объекту `ContactLocation`, который требуется модифицировать.

Во избежание проблемы такого рода можно получать доступ к связанным данным напрямую. В последующих разделах будут представлены два подхода для доступа к связанным данным вместе с объяснениями, когда каждый из них должен использоваться.

Повышение связанных данных

Для данных, играющих важную роль в приложении, вроде тех, что имеют собственные инструменты управления и жизненный цикл, наилучший подход предусматривает их повышение, чтобы к ним можно было обращаться через свойство DbSet<T>, определенное в классе контекста.

Это наиболее разрушительный подход, т.к. он требует создания и применения новой миграции, но он ставит связанные данные в первоклассное положение в рамках приложения и соблюдает соглашения, к которым привыкли многие разработчики.

Чтобы повысить данные Supplier для получения к ним доступа напрямую, добавьте в класс EFDatabaseContext свойство, как показано в листинге 15.2.

Листинг 15.2. Повышение связанных данных в файле EFDatabaseContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace DataApp.Models {
  public class EFDatabaseContext : DbContext {

    public EFDatabaseContext(DbContextOptions<EFDatabaseContext> opts)
      : base(opts) { }

    public DbSet<Product> Products { get; set; }
    public DbSet<Supplier> Suppliers { get; set; }
  }
}
```

Свойство Suppliers возвращает объект DbSet<Supplier>, который может использоваться для запрашивания и оперирования объектами Supplier в БД. При определении свойств DbSet<T> требуется новая миграция, поэтому выполните в папке проекта DataApp команды из листинга 15.3 для создания и применения миграции.

Листинг 15.3. Создание и применение миграции

```
dotnet ef migrations add PromoteSuppliers --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Просмотрев метод Up() в файле <отметка времени>_PromoteSuppliers.cs, который был создан в папке Migrations, вы заметите, что повышение данных Supplier привело к переименованию таблицы БД, содержащей связанные данные:

```
...
migrationBuilder.RenameTable(name: "Supplier", newName: "Suppliers");
...
```

Имя изменилось из-за переключения с одного соглашения Entity Framework Core на другое. В листинге 15.3 соблюдалось соглашение об использовании формы множественного числа для имени свойства (Suppliers), которое инфраструктура Entity Framework Core будет применять в качестве имени таблицы, предназначенной для хранения объектов Supplier в БД. Однако в БД уже присутствует таблица Supplier, которая была создана с учетом соглашения по использованию имени навигационного свойства, определенного в главе 14. Соглашение для DbSet<T> имеет более высокий приоритет и приводит к переименованию таблицы, содержащей объекты Supplier.

Потребление повышенных данных

Доступ к повышенным данным можно осуществлять с применением приемов, описанных в предшествующих главах. Чтобы обеспечить хранилище для объектов `Supplier`, добавьте в папку `Models` файл класса по имени `SupplierRepository.cs` и определите в нем интерфейс и класс, как показано в листинге 15.4.

Листинг 15.4. Содержимое файла `SupplierRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {
  public interface ISupplierRepository {
    Supplier Get(long id);
    IEnumerable<Supplier> GetAll();
    void Create(Supplier newDataObject);
    void Update(Supplier changedDataObject);
    void Delete(long id);
  }
  public class SupplierRepository : ISupplierRepository {
    private EFDatabaseContext context;
    public SupplierRepository(EFDatabaseContext ctx) => context = ctx;
    public Supplier Get(long id) {
      return context.Suppliers.Find(id);
    }
    public IEnumerable<Supplier> GetAll() {
      return context.Suppliers;
    }
    public void Create(Supplier newDataObject) {
      context.Add(newDataObject);
      context.SaveChanges();
    }
    public void Update(Supplier changedDataObject) {
      context.Update(changedDataObject);
      context.SaveChanges();
    }
    public void Delete(long id) {
      context.Remove(Get(id));
      context.SaveChanges();
    }
  }
}
```

В хранилище для объектов `Supplier` используются простые операции без более сложных оптимизаций, описанных в главе 12. Зарегистрируйте хранилище в классе `Startup`, чтобы его можно было потреблять как службу в остальных частях приложения (листинг 15.5).

Листинг 15.5. Регистрация хранилища для повышенных данных в файле `Startup.cs` из папки `DataApp`

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddMvc();
  string conString = Configuration["ConnectionStrings:DefaultConnection"];
  services.AddDbContext<EFDatabaseContext>(options =>
    options.UseSqlServer(conString));

  string customerConString =
    Configuration["ConnectionStrings:CustomerConnection"];
  services.AddDbContext<EFCustomerContext>(options =>
    options.UseSqlServer(customerConString));

  services.AddTransient<IDataRepository, EFDataRepository>();
  services.AddTransient<ICustomerRepository, EFCustomerRepository>();
  services.AddTransient<MigrationsManager>();
  services.AddTransient<ISupplierRepository, SupplierRepository>();
}
...
```

Чтобы предоставить контроллер для данных `Supplier`, добавьте в папку `Controllers` файл класса по имени `RelatedDataController.cs` с определением класса из листинга 15.6.

Листинг 15.6. Содержимое файла `RelatedDataController.cs` из папки `Controllers`

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

namespace DataApp.Controllers {

  public class RelatedDataController : Controller {
    private ISupplierRepository supplierRepo;

    public RelatedDataController(ISupplierRepository repo)
      => supplierRepo = repo;

    public IActionResult Index() => View(supplierRepo.GetAll());
  }
}
```

В контроллере определен единственный метод действия, который запрашивает у хранилища все объекты `Supplier` в БД и передает их стандартному представлению. Для создания этого представления создайте папку `Views/RelatedData` и добавьте в нее файл по имени `Index.cshtml`, содержимое которого приведено в листинге 15.7.

Листинг 15.7. Содержимое файла `Index.cshtml` из папки `Views/RelatedData`

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}
```

```
<table class="table table-striped table-sm">
  <tr><th>ID</th><th>Name</th><th>City</th><th>State</th></tr>
  @foreach (var s in Model.OrderBy(s => s.Id)) {
    <tr>
      <td>@s.Id</td>
      <td>@s.Name</td>
      <td>@s.City</td>
      <td>@s.State</td>
    </tr>
  }
</table>
```

Чтобы увидеть эффект от повышения данных `Supplier`, запустите приложение и перейдите по ссылке `http://localhost:5000/relateddata`; результат показан на рис. 15.2.

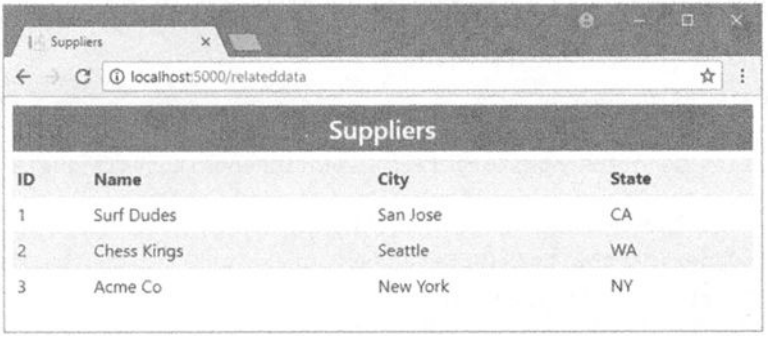

Рис. 15.2. Повышение связанных данных

Доступ к связанным данным с использованием параметра типа

Альтернативой повышению данных является применение набора методов, которые предоставляются классом контекста БД и позволяют указывать тип данных как параметр типа. Поступать так полезно при работе с данными, для которых требуется нерегулярный или ограниченный доступ через специфические операции и повышение до свойства `DbSet<T>` не имеет оснований. В табл. 15.2 описаны методы `DbContext`, которые принимают параметр типа и могут использоваться для доступа к данным без необходимости в их повышении.

Таблица 15.2. Методы `DbContext` с параметрами типа

Имя	Описание
`Set<T>()`	Возвращает объект `DbSet<T>`, который может применяться для запрашивания БД
`Find<T>(key)`	Запрашивает у БД объект типа `T`, который имеет указанный ключ
`Add<T>(newObject)`	Добавляет в БД новый объект типа `T`
`Update<T>(changedObject)`	Обновляет объект типа `T`
`Remove<T>(dataObject)`	Удаляет из БД объект типа `T`

Преимущество методов, перечисленных в табл. 15.2, заключается в том, что они могут использоваться для создания обобщенного хранилища, с помощью которого можно предоставить доступ к специфическому типу, когда хранилище сконфигурировано в качестве службы в классе `Startup`. Добавьте в папку `Models` файл класса по имени `GenericRepository.cs` с определениями интерфейса и класса его реализации, показанными в листинге 15.8.

Листинг 15.8. Содержимое файла `GenericRepository.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {

  public interface IGenericRepository<T> where T : class {

    T Get(long id);

    IEnumerable<T> GetAll();

    void Create(T newDataObject);

    void Update(T changedDataObject);

    void Delete(long id);
  }

  public class GenericRepository<T> : IGenericRepository<T> where T : class
  {
    protected EFDatabaseContext context;

    public GenericRepository(EFDatabaseContext ctx) => context = ctx;

    public virtual T Get(long id) {
      return context.Set<T>().Find(id);
    }

    public virtual IEnumerable<T> GetAll() {
      return context.Set<T>();
    }

    public virtual void Create(T newDataObject) {
      context.Add<T>(newDataObject);
      context.SaveChanges();
    }

    public virtual void Delete(long id) {
      context.Remove<T>(Get(id));
      context.SaveChanges();
    }

    public virtual void Update(T changedDataObject) {
      context.Update<T>(changedDataObject);
      context.SaveChanges();
    }
  }
}
```

В интерфейсе `IGenericRepository<T>` определены операции, которые хранилище обязано предоставлять для работы с типом `T`. Конструкция `where` ограничивает параметр типа классами, что является ограничением, навязываемым инфраструктурой Entity Framework Core.

Класс GenericRepository<T> реализует интерфейс IGenericRepository<T> с помощью тех же самых основных приемов, которые применялись для класса Supplier, но выполняется с использованием методов, описанных в табл. 15.2.

Совет. Методы в листинге 15.8 помечены как virtual, поэтому можно создавать более специализированные реализации класса хранилища без необходимости во внесении обширных изменений в приложение.

Специфические типы, которые будут применяться для интерфейса и класса реализации, конфигурируются при создании службы внедрения зависимостей в классе Startup. Определите службы для классов ContactDetails и ContactLocation (листинг 15.9).

Листинг 15.9. Создание служб в файле Startup.cs из папки DataApp

```
...
public void ConfigureServices(IServiceCollection services) {
  services.AddMvc();
  string conString = Configuration["ConnectionStrings:DefaultConnectio
n"];
  services.AddDbContext<EFDatabaseContext>(options =>
    options.UseSqlServer(conString));

  string customerConString =
    Configuration["ConnectionStrings:CustomerConnection"];
  services.AddDbContext<EFCustomerContext>(options =>
    options.UseSqlServer(customerConString));

  services.AddTransient<IDataRepository, EFDataRepository>();
  services.AddTransient<ICustomerRepository, EFCustomerRepository>();
  services.AddTransient<MigrationsManager>();
  services.AddTransient<ISupplierRepository, SupplierRepository>();
  services.AddTransient<IGenericRepository<ContactDetails>,
    GenericRepository<ContactDetails>>();
  services.AddTransient<IGenericRepository<ContactLocation>,
    GenericRepository<ContactLocation>>();
}
...
```

Чтобы отобразить данные, полученные через обобщенные методы контекста, добавьте в контроллер RelatedData методы действий, как показано в листинге 15.10. Эти методы запрашивают все доступные объекты каждого типа, хотя можно использовать все стандартные операции над данными.

Листинг 15.10. Запрашивание данных в файле RelatedDataController.cs из папки Controllers

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

namespace DataApp.Controllers {

  public class RelatedDataController : Controller {
    private ISupplierRepository supplierRepo;
```

```
    private IGenericRepository<ContactDetails> detailsRepo;
    private IGenericRepository<ContactLocation> locsRepo;

    public RelatedDataController(ISupplierRepository sRepo,
        IGenericRepository<ContactDetails> dRepo,
        IGenericRepository<ContactLocation> lRepo) {
      supplierRepo = sRepo;
      detailsRepo = dRepo;
      locsRepo = lRepo;
    }

    public IActionResult Index() => View(supplierRepo.GetAll());

    public IActionResult Contacts() => View(detailsRepo.GetAll());
    public IActionResult Locations() => View(locsRepo.GetAll());
  }
}
```

Внимание! У вас может возникнуть соблазн развить идею доступа к классам данных с применением параметров типа и настроить в конвейере ASP.NET Core обобщенный обработчик и контроллер, не требующий методов действий и представлений для каждого класса. Решение может показаться удачным, но его следует избегать, поскольку оно открывает доступ ко всем данным в БД, что редко считается приемлемым. Работайте с параметрами типа "за кулисами", чтобы добиться простоты приложения, но явно предоставляйте доступ к классам на индивидуальной основе.

Обобщенные хранилища принимаются через конструктор обычным способом и используются методами действий `Contacts()` и `Locations()`, которые оба применяют стандартное представление. Чтобы отобразить объекты `ContactDetails`, создайте в папке `Views/RelatedData` файл по имени `Contacts.cshtml` с содержимым из листинга 15.11.

Листинг 15.11. Содержимое файла `Contacts.cshtml` из папки `Views/RelatedData`

```
@model IEnumerable<DataApp.Models.ContactDetails>
@{
  ViewData["Title"] = "ContactDetails";
  Layout = "_Layout";
}
<table class="table table-striped table-sm">
  <tr><th>ID</th><th>Name</th><th>Phone</th></tr>
  @foreach (var s in Model) {
    <tr><td>@s.Id</td><td>@s.Name</td><td>@s.Phone</td></tr>
  }
</table>
```

Представление отображает свойства `ContactDetails` в таблице. Чтобы обеспечить подходящее представление для объектов `ContactLocation`, добавьте в папку `Views/RelatedData` файл по имени `Locations.cshtml`, содержимое которого приведено в листинге 15.12.

Листинг 15.12. Содержимое файла `Locations.cshtml` из папки `Views/RelatedData`

```
@model IEnumerable<DataApp.Models.ContactLocation>
@{
  ViewData["Title"] = "ContactLocations";
  Layout = "_Layout";
}
<table class="table table-striped table-sm">
  <tr><th>ID</th><th>Name</th></tr>
  @foreach (var s in Model) {
    <tr><td>@s.Id</td><td>@s.LocationName</td></tr>
  }
</table>
```

Для доступа к связанным данным посредством обобщенного хранилища запустите приложение и перейдите по ссылкам `http://localhost:5000/relateddata/contacts` и `http://localhost:5000/relateddata/locations`, которые нацелены на методы действий, определенные в листинге 15.10; результат можно видеть на рис. 15.3.

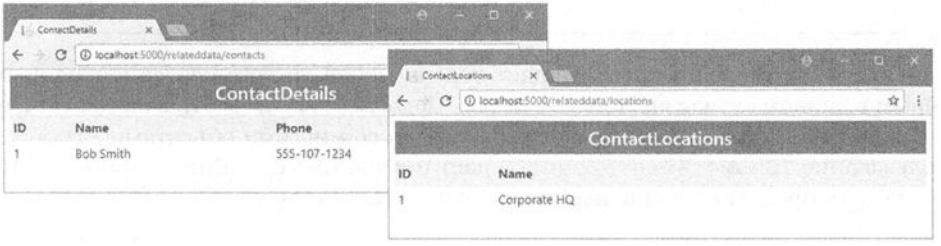

Рис. 15.3. Доступ к связанным данным через обобщенное хранилище

Укомплектование отношения между данными

Повышение связанных данных облегчило доступ к ним, но работу с ними можно продолжить совершенствовать. В настоящий момент отношения между классами протекают только в одном направлении. Например, можно начать с объекта `Product` и проследовать по навигационному свойству, чтобы получить связанные данные `Supplier`, но начать с объекта `Supplier` и двигаться в противоположном направлении нельзя. Для устранения указанного недостатка инфраструктура Entity Framework Core позволяет определять свойства, которые делают возможной навигацию в другом направлении, что известно как *укомплектование отношения*.

Существует ряд различных типов отношений, которые можно создавать между классами, но когда впервые определяется навигационное свойство, инфраструктура Entity Framework Core предполагает, что нужно создать отношение "один ко многим", и навигационное свойство добавляется в класс, находящийся на стороне "многие" отношения. Именно это произошло при добавлении навигационного свойства к классу `Product` в главе 14: инфраструктура Entity Framework Core создала отношение "один ко многим", которое позволило каждому объекту `Supplier` быть связанным со многими объектами `Product`, и соответствующим образом сконфигурировала БД.

Совет. В главе 16 объясняется, как создавать другие типы отношений.

Укомплектование отношения "один ко многим" делается легко и требует добавления навигационного свойства в класс, находящийся на стороне "один" отношения. Такое свойство называется *обратным*. Чтобы укомплектовать отношение между классами `Product` и `Supplier`, добавьте в класс `Supplier` обратное свойство, как показано в листинге 15.13.

Листинг 15.13. Укомплектование отношения в файле `Supplier.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {

  public class Supplier {
    public long Id { get; set; }
    public string Name { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public ContactDetails Contact { get; set; }
    public IEnumerable<Product> Products { get; set; }
  }
}
```

Навигационное свойство возвращает перечисление другого класса в отношении, которым в данном случае является `Product`. Подобным образом отражается тот факт, что каждый объект `Supplier` может быть связан со многими объектами `Product`, и использование `IEnumerable<Product>` дает инфраструктуре Entity Framework Core возможность предоставить полный набор связанных данных.

На заметку! При укомплектовании отношения "один ко многим" создавать новую миграцию не нужно. Новая миграция требуется для других типов отношений, как демонстрируется в главе 16.

Запрашивание связанных данных в отношении "один ко многим"

После укомплектования отношения можно начинать с объекта любого типа в отношении и двигаться в обоих направлениях. В примере приложения это означает, что можно начать с объекта `Supplier` и проследовать по свойству `Products`, чтобы переместиться на набор связанных объектов `Product` при условии включения связанных данных в запрос. Включить связанные данные в укомплектованном отношении "один ко многим" можно несколькими способами, которые объясняются в последующих разделах.

Запрашивание всех связанных данных

Если необходим полный набор связанных данных, тогда можно применить метод `Include()` или `ThenInclude()` для расширения запроса путем выбора навигационного свойства. В листинге 15.14 с помощью метода `Include()` обеспечивается прослеживание по свойству `Supplier.Products`, определенному в листинге 15.13, и запрашиваются все объекты `Product`, которые связаны с каждым объектом `Supplier`.

Листинг 15.14. Запрашивание всех данных в файле SupplierRepository.cs из папки Models

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

namespace DataApp.Models {
  public interface ISupplierRepository {
    Supplier Get(long id);
    IEnumerable<Supplier> GetAll();
    void Create(Supplier newDataObject);
    void Update(Supplier changedDataObject);
    void Delete(long id);
  }
  public class SupplierRepository : ISupplierRepository {
    private EFDatabaseContext context;
    public SupplierRepository(EFDatabaseContext ctx) => context = ctx;
    public Supplier Get(long id) {
      return context.Suppliers.Find(id);
    }
    public IEnumerable<Supplier> GetAll() {
      return context.Suppliers.Include(s => s.Products);
    }
    public void Create(Supplier newDataObject) {
      context.Add(newDataObject);
      context.SaveChanges();
    }
    public void Update(Supplier changedDataObject) {
      context.Update(changedDataObject);
      context.SaveChanges();
    }
    public void Delete(long id) {
      context.Remove(Get(id));
      context.SaveChanges();
    }
  }
}
```

Изменения в хранилище сообщают инфраструктуре Entity Framework Core о необходимости запрашивания всех объектов `Product`, ассоциированных с объектом `Supplier`. Для отображения связанных данных пользователю и построения набора операций над данными потребуется внести дополнительные изменения. Чтобы отобразить объекты `Product`, связанные с объектом `Supplier`, добавьте содержимое листинга 15.15 в файл `Index.cshtml` из папки `Views/RelatedData`.

Листинг 15.15. Отображение связанных данных в файле Index.cshtml из папки Views/RelatedData

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}
```

```
<table class="table table-striped table-sm">
  <tr><th>ID</th><th>Name</th><th>City</th><th>State</th></tr>
  @foreach (var s in Model.OrderBy(s => s.Id)) {
    <tr>
      <td>@s.Id</td>
      <td>@s.Name</td>
      <td>@s.City</td>
      <td>@s.State</td>
    </tr>
    @if (s.Products != null) {
      @foreach (var p in s.Products) {
        <tr class="table-dark">
          <td></td>
          <td>@p.Name</td>
          <td>@p.Category</td>
          <td>@p.Price</td>
        </tr>
      }
    }
  }
</table>
```

Внесенные в представление изменения отображают сводку по каждому объекту `Product` после объекта `Supplier`, с которым он связан. Чтобы увидеть эффект, запустите приложение и перейдите по ссылке `http://localhost:5000/relateddata`. Объекты `Product` находятся в строках, отображенных более темным цветом (рис. 15.4).

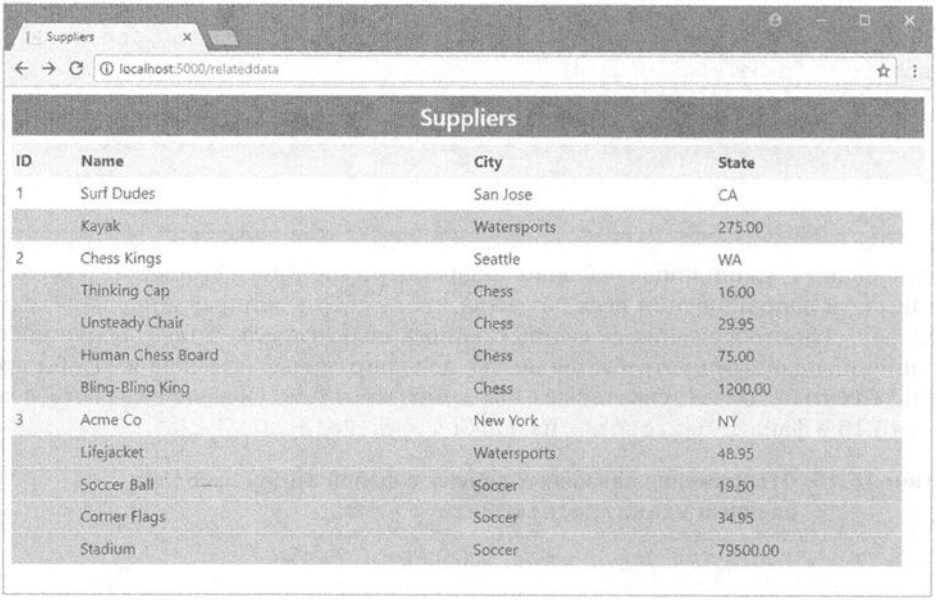

Рис. 15.4. Запрашивание всех связанных данных в отношении "один ко многим"

При рассмотрении различных способов, которыми можно получить связанные данные в отношении "один ко многим", полезно исследовать запрос, используемый для извлечения данных из БД. В журнальных сообщениях, сгенерированных приложением, вы заметите, что первыми запрашиваются данные Supplier:

```
...
SELECT [s].[Id], [s].[City], [s].[ContactId], [s].[Name], [s].[State]
FROM [Suppliers] AS [s]
ORDER BY [s].[Id]
...
```

Для получения связанных данных посылается второй запрос, который извлекает связанные данные Product:

```
...
SELECT [s.Products].[Id], [s.Products].[Category], [s.Products].[Color],
  [s.Products].[InStock], [s.Products].[Name], [s.Products].[Price],
  [s.Products].[SupplierId]
FROM [Products] AS [s.Products]
INNER JOIN (
  SELECT [s0].[Id]
  FROM [Suppliers] AS [s0]
) AS [t] ON [s.Products].[SupplierId] = [t].[Id]
ORDER BY [t].[Id]
...
```

Запрашивание с использованием явной загрузки

Применение метода Include() для следования по навигационному свойству в отношении "один ко многим" не предлагает какого-нибудь способа фильтрации связанных данных, т.е. все они извлекаются из БД. Если нужна большая избирательность, тогда инфраструктура Entity Framework Core предоставляет два приема фильтрации запрашиваемых данных. Первый прием называется *явной загрузкой*, которая в листинге 15.16 используется взамен метода Include().

Листинг 15.16. Применение явной загрузки в файле SupplierRepository.cs из папки Models

```
...
public IEnumerable<Supplier> GetAll() {
  IEnumerable<Supplier> data = context.Suppliers.ToArray();
  foreach (Supplier s in data) {
    context.Entry(s).Collection(e => e.Products)
      .Query()
      .Where(p => p.Price > 50)
      .Load();
  }
  return data;
}
...
```

Явная загрузка опирается на метод DbContext.Entry(), который применялся в главе 12 для доступа к данным отслеживания изменений. Вызов метода Entry(), определенного в объекте контекста, возвращает объект EntityEntry, который предоставляет два метода для доступа к связанным данным, описанные в табл. 15.3.

Таблица 15.3. Методы объекта `EntityEntry`, предназначенные для доступа к связанным данным

Имя	Описание
`Reference(name)`	Используется для навигационных свойств, которые нацелены на одиночный объект, указанный либо как строка, либо с помощью лямбда-выражения для выбора свойства
`Collection(name)`	Применяется для навигационных свойств, которые нацелены на коллекцию, указанную либо как строка, либо с помощью лямбда-выражения для выбора свойства

После использования метода `Reference()` или `Collection()` для выбора навигационного свойства вызывается метод `Query()`, чтобы получить объект `IQueryable`, который может применяться с LINQ для фильтрации загружаемых данных. В листинге 15.16 метод `Where()` используется для фильтрации связанных данных на основе цены, сообщая инфраструктуре Entity Framework Core о запрашивании только тех связанных объектов `Product`, значение свойства `Price` у которых превышает 50.

Метод `Load()` применяется для принудительного выполнения запроса. Обычно это не требуется, потому что запрос будет выполняться автоматически, когда объекты `IQueryable<T>` перечисляется представлением Razor или методом LINQ. Тем не менее, в данном случае вызов метода `Load()` необходим, поскольку объекты `IQueryable<T>`, возвращенные методом `Query()`, никогда не перечисляются. Без вызова метода `Load()` запросы связанных данных не выполнились бы, и представлению Razor передавались бы только объекты `Supplier`. (Метод `Load()` дает тот же самый эффект, что и `ToArray()` или `ToList()`, но не создает и не возвращает коллекцию объектов.)

На заметку! В листинге 15.6 обратите внимание на использование метода `ToArray()` перед перечислением объектов `Supplier` в цикле `foreach`. Метод `ToArray()` инициирует выполнение запроса, который получает данные `Supplier`, избегая ситуации, когда один запрос выполняется при перечислении объектов `IQueryable<Supplier>` циклом `foreach` в методе хранилища и при запуске дублированного запроса циклом `foreach` в представлении Razor.

Чтобы увидеть эффект от применения явной загрузки, перезапустите приложение и перейдите по ссылке `http://localhost:5000/relateddata`. Отображаемые данные `Product` содержат только отфильтрованные объекты (рис. 15.5).

Недостаток применения явной загрузки в том, что она генерирует много запросов к БД. Если вы просмотрите журнальные сообщения, сгенерированные приложением, то заметите первый запрос для данных `Supplier`:

```
...
SELECT [s].[Id], [s].[City], [s].[ContactId], [s].[Name], [s].[State]
FROM [Suppliers] AS [s]
...
```

Так как каждый объект `Supplier` обрабатывается циклом `foreach` и связанные данные загружаются явно, в БД отправляется еще один запрос:

```
...
SELECT [e].[Id], [e].[Category], [e].[Color], [e].[InStock], [e].[Name],
  [e].[Price], [e].[SupplierId]
FROM [Products] AS [e]
WHERE ([e].[SupplierId] = @__get_Item_0) AND ([e].[Price] > 50.0)
...
```

В БД хранятся три объекта Supplier, а это значит, что для получения данных в приведенном примере понадобится в общей сложности четыре запроса. Количество запросов может увеличиваться для большого количества объектов, так что прием явной загрузки наиболее эффективен при наличии малого числа объектов, которым требуются связанные данные. В случае работы с крупными количествами объектов может больше подойти прием, описанный в следующем разделе.

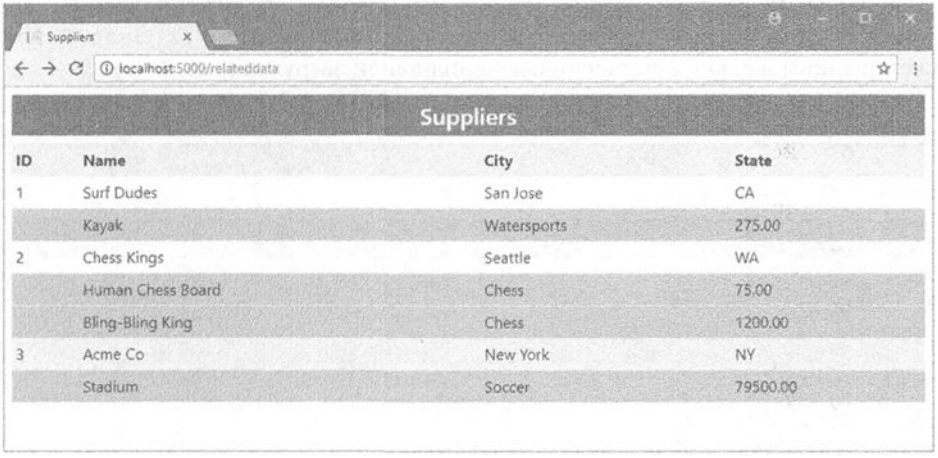

Рис. 15.5. Использование явной загрузки для фильтрации данных

Запрашивание с использованием средства исправления

Инфраструктура Entity Framework Core поддерживает средство, называемое *исправлением* (fixing up), когда данные, извлеченные из объекта контекста БД, кешируются и применяются для заполнения навигационных свойств объектов, которые создаются в последующих запросах. При осмотрительном использовании это средство позволяет создавать сложные запросы, которые получают связанные данные более эффективно, чем явная загрузка или следование по навигационному свойству с применением метода Include(). В листинге 15.17 на основе средства исправления получаются объекты Supplier и связанные с ними объекты Product, значение свойства Price которых превышает 50.

Листинг 15.17. Использование средства исправления в файле SupplierRepository.cs из папки Models

```
...
public IEnumerable<Supplier> GetAll() {
  context.Products.Where(p => p.Supplier != null && p.Price > 50).Load();
  return context.Suppliers;
}
...
```

Для приведенного в листинге 15.17 примера требуются два запроса. Первый запрос применяет свойство Products объекта контекста, чтобы извлечь все объекты Product, связанные с объектом Supplier, у которых свойство Price имеет значе-

ние больше 50. Единственная цель этого запроса — заполнить кеш Entity Framework Core объектами данных, а потому использовался метод Load() для принудительной оценки запроса.

Второй запрос применяет свойство Suppliers для извлечения объектов Supplier. Запрос будет выполняться при перечислении последовательности объектов в представлении Razor, так что вызов метода Load() не требуется. Когда выполняется второй запрос, инфраструктура Entity Framework Core автоматически проверит данные, кешированные из первого запроса, и будет использовать их для заполнения навигационных свойств Supplier.Products подходящими объектами Product. Результат окажется таким же, как в случае применения явной загрузки (рис. 15.6).

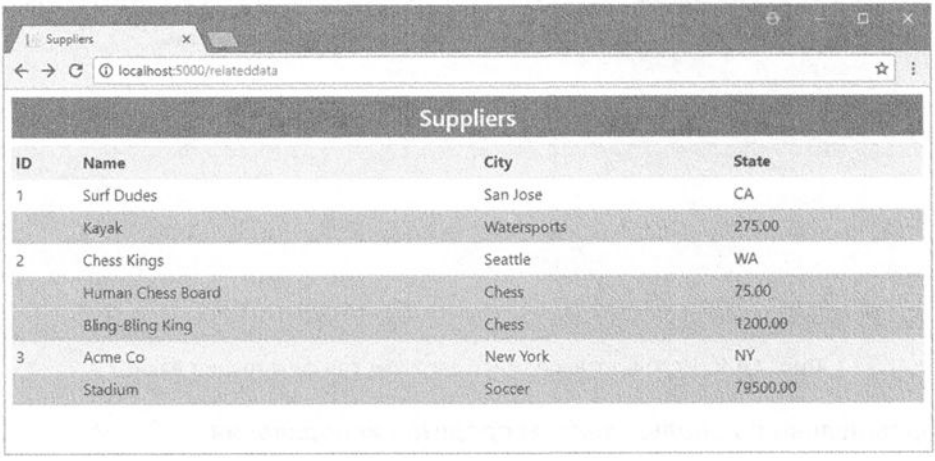

Рис. 15.6. Использование средства исправления для фильтрации данных

Внимание! Применение средства исправления может быть сопряжено с рядом проб и ошибок и требует вдумчивого отношения к запросам, которые отправляются БД. Можно получить все требующиеся данные с меньшим числом запросов, чем при явной загрузке, но удивительно легко не достигнуть своей цели и либо запрашивать чересчур много или слишком мало данных, либо генерировать больше запросов, чем предполагалось.

Просмотрев журнальные сообщения, выданные приложением, вы увидите запросы, которые были отправлены БД. Первый запрос извлекает объекты Product, которые удовлетворяют критерию, указанному в методе Where():

```
...
SELECT [p].[Id], [p].[Category], [p].[Color], [p].[InStock], [p].[Name],
  [p].[Price], [p].[SupplierId]
FROM [Products] AS [p]
WHERE [p].[SupplierId] IS NOT NULL AND ([p].[Price] > 50.0)
...
```

Второй запрос извлекает все доступные данные Supplier:

```
...
SELECT [s].[Id], [s].[City], [s].[ContactId], [s].[Name], [s].[State]
FROM [Suppliers] AS [s]
...
```

Процесс исправления использует данные из первого запроса для заполнения навигационных свойств объектов, созданных вторым запросом. Такой полностью автоматический процесс будет выполняться в случае применения одного и того же объекта контекста БД для выполнения множества запросов (одна из причин соблюдения осторожности в отношении жизненного цикла объектов хранилища, сконфигурированного в классе `Startup`, чтобы не получать исправленные данные, когда они не ожидаются).

Ловушка, скрываемая средством исправления

Возможность исправления означает, что инфраструктура Entity Framework Core будет заполнять навигационные свойства объектами, которые создала ранее. При аккуратном использовании процесс исправления является мощным инструментом для выбора только требующихся данных, не запрашивая бесконечно БД. Но процесс исправления не может быть отключен и скрывает в себе ловушку, в которую можно попасть по неосторожности, если следовать по навигационным свойствам, не принимая во внимание выполненные запросы.

Для демонстрации проблемы создайте в папке `Views/Home` представление по имени `SupplierRelated.cshtml` и поместите в него содержимое листинга 15.18, чтобы отобразить пользователю объект `Supplier` и его список объектов `Product`.

Листинг 15.18. Содержимое файла `SupplierRelated.cshtml` из папки `Views/Home`

```
@model DataApp.Models.Supplier

@if (Model?.Products == null) {
  <tr><td colspan="6" class="text-center table-dark">No Related Data</td>
  </tr>
} else {
  @foreach (Product p in Model?.Products) {
    <tr class="table-dark">
      <td colspan="3"></td>
      <td>@p.Name</td>
      <td>@p.Category</td>
      <td>@p.Price</td>
    </tr>
  }
}
```

Представление получает объект `Supplier` в качестве своей модели и следует по навигационному свойству `Products` для создания простой таблицы связанных объектов `Product`. Чтобы задействовать это представление, внесите изменение, показанное в листинге 15.19, в файл `Index.cshtml` из папки `Views/Home`.

Листинг 15.19. Отображение связанных данных в файле `Index.chstml` из папки `Views/Home`

```
...
<tbody>
  @foreach (var p in Model) {
    <tr>
      <td>@p.Id</td><td>@p.Name</td><td>@p.Category</td>
      <td>$@p.Price.ToString("F2")</td>
```

```
      @if (ViewBag.includeRelated) {
        <td>@p.Supplier?.Name</td>
      }
      <td>
        <form asp-action="Delete" method="post">
          <a asp-action="Edit" class="btn btn-sm btn-warning"
             asp-route-id="@p.Id">
            Edit
          </a>
          <input type="hidden" name="id" value="@p.Id" />
          <button type="submit" class="btn btn-danger btn-sm">
            Delete
          </button>
        </form>
      </td>
    </tr>
    <tr>
      <td colspan="6">@Html.Partial("SupplierRelated", p.Supplier)</td>
    </tr>
  }
</tbody>
...
```

Для выявления проблемы перезапустите приложение и перейдите по ссылке `http://localhost:5000`. Процесс исправления применяется для заполнения навигационных свойств, когда инфраструктура Entity Framework Core обрабатывает ответ, приводя к несогласованности данных (рис. 15.7).

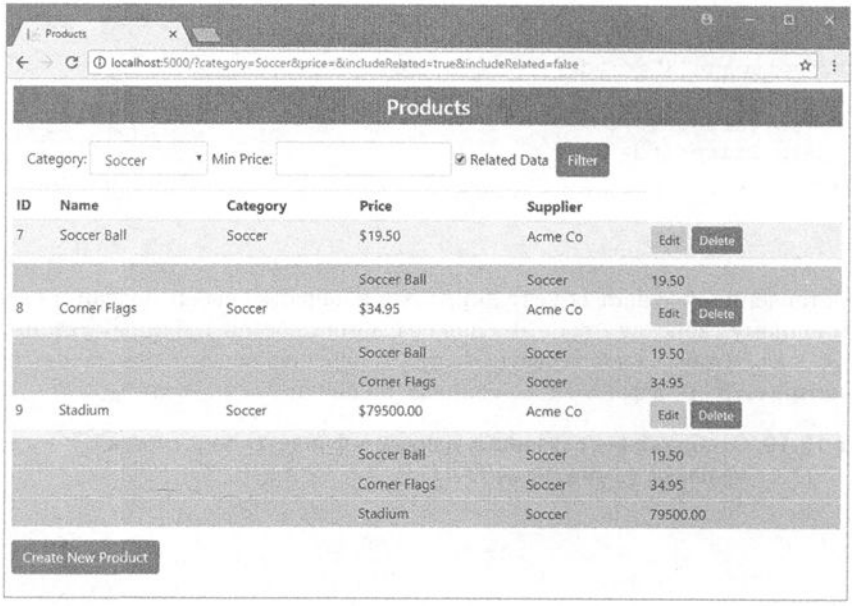

Рис. 15.7. Переход к данным, выходящим за рамки запроса

На рис. 15.7 отображается свойство Products в категории Soccer и видно, что для каждого элемента связанные данные отличаются. Так произошло из-за следования по навигационным свойствам за рамками связанных данных, указанных в запросе, начиная с объекта Product, переходя на связанный объект Supplier и затем возвращаясь обратно к связанным объектам Product через свойство Products. Инфраструктура Entity Framework Core использует исправленные данные для заполнения свойства Supplier.Products, но располагает только теми объектами Product, которые были созданы до сих пор, а их число растет с каждым новым обработанным объектом Product, давая несогласованные результаты, что проиллюстрировано на рис. 15.7. Чтобы избежать проблемы подобного рода, не следует переходить по свойствам, которые не выбирались с помощью метода Include() или ThenInclude() либо исправлялись с применением предыдущего запроса.

Работа со связанными данными в отношении "один ко многим"

Одной из наиболее важных характеристик укомплектованных отношений является то, что операции могут выполняться с использованием навигационных свойств на обеих сторонах отношения. В последующих разделах будет показано, как применять навигационные свойства в отношении "один ко многим" между классами Supplier и Product для выполнения различных операций.

В качестве подготовки измените запрос, используемый для объектов Supplier в классе SupplierRepository, чтобы включить в него все связанные объекты Product (листинг 15.20).

Листинг 15.20. Изменение запроса в файле SupplierRepository.cs из папки Models

```
...
public IEnumerable<Supplier> GetAll() {
  return context.Suppliers.Include(p => p.Products);
}
...
```

Для отделения текущего примера от других примеров в настоящей главе добавьте в папку Controllers файл класса по имени SuppliersController.cs и определите в нем контроллер, как показано в листинге 15.21.

Листинг 15.21. Содержимое файла SuppliersController.cs из папки Controllers

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

namespace DataApp.Controllers {

  public class SuppliersController : Controller {
    private ISupplierRepository supplierRepository;

    public SuppliersController(ISupplierRepository supplierRepo) {
      supplierRepository = supplierRepo;
    }
```

```
    public IActionResult Index() {
      return View(supplierRepository.GetAll());
    }
  }
}
```

В контроллере определен единственный метод действия, который передает все объекты `Supplier` стандартному представлению. Чтобы отобразить объекты `Supplier` и связанные объекты `Product`, создайте папку `Views/Suppliers` и добавьте в нее файл по имени `Index.cshtml` с содержимым из листинга 15.22.

Листинг 15.22. Содержимое файла `Index.cshtml` из папки `Views/Suppliers`

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}
@foreach (Supplier s in Model) {
  <h4 class="bg-info text-center text-white p-1">@s.Name</h4>
  <div class="container-fluid">
    @if (s.Products == null || s.Products.Count() == 0) {
      <div class="p-1 text-center">No Products</div>
    } else {
      @foreach (Product p in s.Products) {
        <div class="row p-1">
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col">@p.Price</div>
        </div>
      }
    }
  </div>
}
```

Для просмотра содержимого, сгенерированного представлением, перезапустите приложение и перейдите по ссылке `http://localhost:5000/suppliers`. Отобразятся названия всех поставщиков вместе с деталями связанных с ними товаров (рис. 15.8).

Обновление связанных объектов

Ранее в главе было показано, как редактировать значения объектов `Product`, которые извлекались из БД напрямую с применением свойства `DbSet<Product>`, определенного в классе контекста БД. Это самый простой и прямой способ выполнения обновлений, но вносить изменения также можно, начиная с объекта `Supplier` и получая доступ к нужным объектам `Product` через навигационное свойство. Чтобы продемонстрировать редактирование через навигационное свойство, создайте в папке `Views/Suppliers` представление по имени `Editor.cshtml` и поместите в него содержимое, приведенное в листинге 15.23.

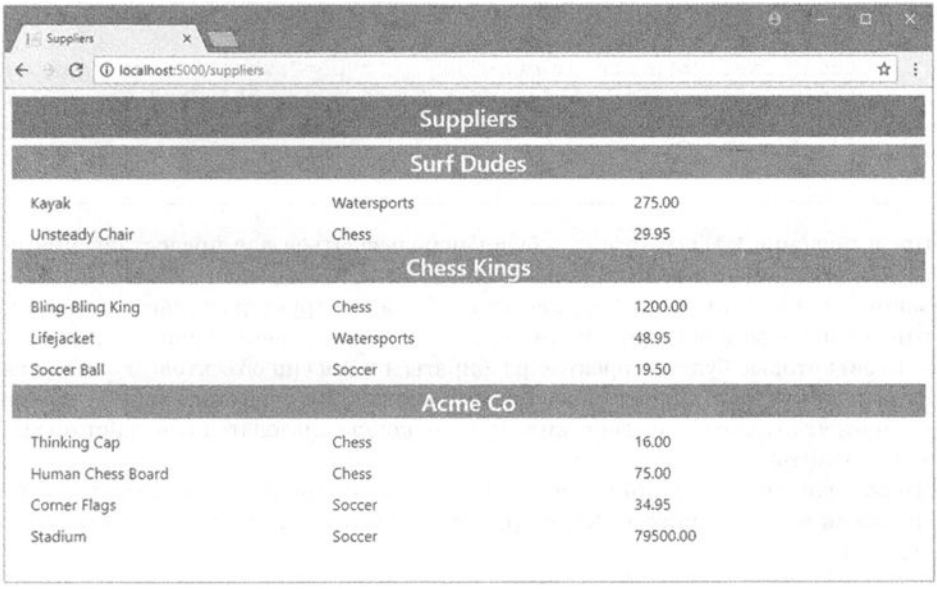

Рис. 15.8. Отображение объектов `Supplier` и `Product`

Листинг 15.23. Содержимое файла `Editor.cshtml` из папки `Views/Suppliers`

```
@model Supplier
@{
  int counter = 0;
}

<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="Name" />
  <input type="hidden" asp-for="City" />
  <input type="hidden" asp-for="State" />

  @foreach (Product p in Model.Products) {
    <div class="row">
      <input type="hidden" name="Products[@counter].Id" value="@p.Id" />
      <div class="col">
        <input name="Products[@counter].Name" value="@p.Name"
              class="form-control"/>
      </div>
      <div class="col">
        <input name="Products[@counter].Category" value="@p.Category"
              class="form-control" />
      </div>
      <div class="col">
        <input name="Products[@counter].Price" value="@p.Price"
              class="form-control"/>
      </div>
      @{ counter++; }
    </div>
  }
```

```
  <div class="row">
    <div class="col text-center m-1">
      <button class="btn btn-sm btn-danger" type="submit">Save</button>
      <a class="btn btn-sm btn-secondary" asp-action="Index">Cancel</a>
    </div>
  </div>
</form>
```

Представление `Editor.cshtml` будет использоваться для предоставления пользователю возможности редактировать значения свойств у всех объектов `Product`, связанных с объектом `Supplier`. Слегка неуклюжая структура полагается на инкрементирование переменной `int` по имени `counter`, применяемой при создании HTML-элементов, которые будут корректно разбираться в массив объектов `Product` связывателем моделей MVC. Значения свойств отображаются с помощью сетки элементов `input` и будут отправляться действию `Update`, когда пользователь щелкнет на кнопке Save (Сохранить).

Чтобы включить представление `Editor` в состав приложения, добавьте элементы, показанные в листинге 15.24, в представление `Index.cshtml` из папки `Views/Suppliers`.

Листинг 15.24. Встраивание представления `Editor` в файле `Index.cshtml` из папки `Views/Suppliers`

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}

@foreach (Supplier s in Model) {
  <h4 class="bg-info text-center text-white p-1">
    @s.Name
    <a asp-action="Edit" asp-route-id="@s.Id"
       class="btn btn-sm btn-warning">
      Edit
    </a>
  </h4>
  <div class="container-fluid">
    @if (s.Products == null || s.Products.Count() == 0) {
      <div class="p-1 text-center">No Products</div>
    } else if (ViewBag.SupplierEditId == s.Id) {
      @Html.Partial("Editor", s);
    } else {
      @foreach (Product p in s.Products) {
        <div class="row p-1">
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col">@p.Price</div>
        </div>
      }
    }
  </div>
}
```

Первое добавление помещает рядом с названием каждого поставщика кнопку Edit (Редактировать), которая нацелена на действие Edit и включает свойство Id объекта Supplier. Щелчок на кнопке Edit запускает процесс редактирования путем установки свойства SupplierEditId в объекте ViewBag, которое используется вторым добавлением в листинге 15.24 для отображения представления Editor.

Для обеспечения поддержки, требуемой представлениями, добавьте в контроллер Suppliers методы действий из листинга 15.25.

Листинг 15.25. Добавление действий в файле SuppliersController.cs из папки Controllers

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

namespace DataApp.Controllers {

  public class SuppliersController : Controller {
    private ISupplierRepository supplierRepository;

    public SuppliersController(ISupplierRepository supplierRepo) {
      supplierRepository = supplierRepo;
    }

    public IActionResult Index() {
      ViewBag.SupplierEditId = TempData["SupplierEditId"];
      return View(supplierRepository.GetAll());
    }

    public IActionResult Edit(long id) {
      TempData["SupplierEditId"] = id;
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Update(Supplier supplier) {
      supplierRepository.Update(supplier);
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы увидеть, как работает процесс редактирования, перезапустите приложение, перейдите по ссылке http://localhost:5000/suppliers и щелкните на кнопке Edit для поставщика Chess Kings. Внесите изменения с помощью элементов input и щелкните на кнопке Save для обновления БД (рис. 15.9).

После щелчка на кнопке Save браузер посылает HTTP-запрос, содержащий значения, которые требуются связывателю моделей MVC для создания объекта Supplier и коллекции объектов Product. Связыватель моделей MVC автоматически присваивает коллекцию объектов Product свойству Supplier.Products, которое затем применяется для обновления БД. Инфраструктура Entity Framework Core не может отслеживать изменения в этих объектах (т.к. они были созданы связывателем моделей MVC), а потому объект Supplier должен включать значения для всех своих свойств, что и делалось в представлении Editor.

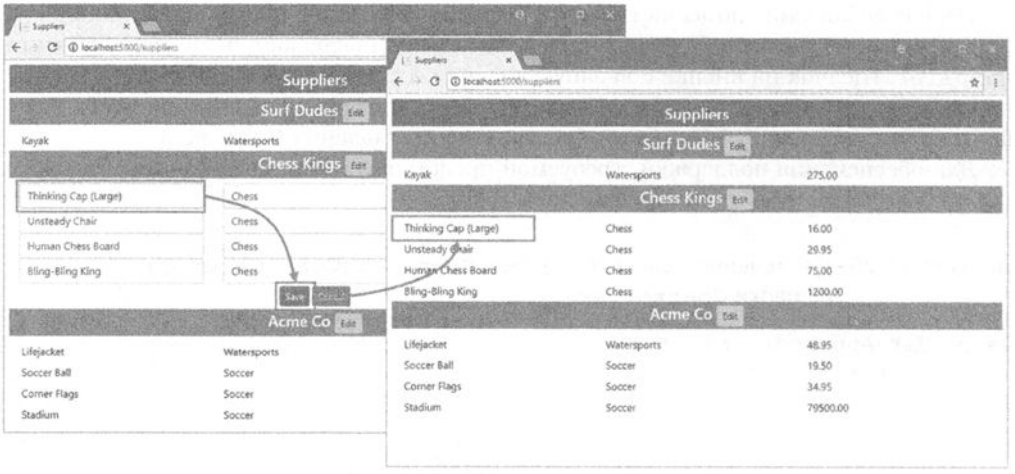

Рис. 15.9. Редактирование значений свойств через навигационное свойство

На заметку! Имейте в виду, что укомплектование отношения не препятствует работе с объектом напрямую. Например, укомплектование отношения `Product/Supplier` вовсе не означает, что объекты `Product` можно обновлять только через навигационное свойство `Supplier.Products`; объекты `Product` по-прежнему разрешено запрашивать напрямую и обновлять их по отдельности. Укомплектование отношения открывает новые способы работы и позволяет выбирать из них тот, который наиболее удобен для проекта.

Создание новых связанных объектов

Описанный в предыдущем разделе прием можно легко адаптировать для создания новых связанных объектов. Когда инфраструктура Entity Framework Core обрабатывает коллекцию объектов `Product`, которые были созданы связывателем моделей MVC, любой объект `Product` с нулевым значением свойства `Id` будет добавлен в БД как новый объект. Это удобный способ создания новых объектов, которые автоматически ассоциируются с объектом `Supplier`, и полезный метод создания новых объектов, не нарушая обязательное ограничение отношения между объектами `Product` и `Supplier`.

Чтобы добавить поддержку для создания нового объекта `Product`, добавьте в папку `Views/Suppliers` представление по имени `Create.cshtml` с содержимым, показанным в листинге 15.26.

Листинг 15.26. Содержимое файла `Create.cshtml` из папки `Views/Suppliers`

```
@model Supplier
@{
  int counter = 0;
}

<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="Name" />
  <input type="hidden" asp-for="City" />
  <input type="hidden" asp-for="State" />
```

```
  @foreach (Product p in Model.Products) {
    <input type="hidden" name="Products[@counter].Id" value="@p.Id" />
    <input type="hidden" name="Products[@counter].Name" value="@p.Name" />
    <input type="hidden" name="Products[@counter].Category"
          value="@p.Category" />
    <input type="hidden" name="Products[@counter].Price" value="@p.Price" />
    counter++;
  }
  <div class="row">
    <div class="col">
      <input name="Products[@counter].Name" value="" class="form-control" />
    </div>
    <div class="col">
      <input name="Products[@counter].Category" class="form-control" />
    </div>
    <div class="col">
      <input name="Products[@counter].Price" class="form-control" />
    </div>
  </div>
  <div class="row">
    <div class="col text-center m-1">
      <button class="btn btn-sm btn-danger" type="submit">Save</button>
      <a class="btn btn-sm btn-secondary" asp-action="Index">Cancel</a>
    </div>
  </div>
</form>
```

При обновлении данных через навигационное свойство вы должны позаботиться о включении в HTML-форму всех существующих объектов. Например, когда инфраструктура Entity Framework Core принимает коллекцию объектов `Product`, она предполагает, что получен полный набор, и попытается нарушить отношения с любым объектом `Product`, который не находится в коллекции. Как следствие, вы обязаны включать значения данных формы для всех существующих данных, а также создаваемых элементов, чтобы пользователь мог ввести новые значения, что и делалось в листинге 15.26.

Для интегрирования нового представления в приложение добавьте в файл `Index.cshtml` из папки `Views/Suppliers` элементы, приведенные в листинге 15.27.

Листинг 15.27. Использование частичного представления в файле `Index.cshtml` из папки `Views/Suppliers`

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}

@foreach (Supplier s in Model) {
  <h4 class="bg-info text-center text-white p-1">
    @s.Name
    <a asp-action="Edit" asp-route-id="@s.Id"
       class="btn btn-sm btn-warning">Edit</a>
```

```
    <a asp-action="Create" asp-route-id="@s.Id"
      class="btn btn-sm btn-danger">Add</a>
  </h4>
  <div class="container-fluid">
    @if (s.Products == null || s.Products.Count() == 0) {
      <div class="p-1 text-center">No Products</div>
    } else if (ViewBag.SupplierEditId == s.Id) {
      @Html.Partial("Editor", s);
    } else {
      @foreach (Product p in s.Products) {
        <div class="row p-1">
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col">@p.Price</div>
        </div>
      }
      if (ViewBag.SupplierCreateId == s.Id) {
        @Html.Partial("Create", s);
      }
    }
  </div>
}
```

Кнопка Add (Добавить) посылает запрос действию `Create`, и содержимое представления `Create.cshtml` отобразится, когда объект `ViewBag` содержит свойство `SupplierCreateId`, значение которого соответствует свойству `Id` обрабатываемого объекта `Supplier`. Для поддержки новых возможностей представления добавьте в контроллер `Suppliers` метод действия из листинга 15.28.

Листинг 15.28. Добавление действия в файле `SuppliersController.cshtml` из папки `Controllers`

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

namespace DataApp.Controllers {

  public class SuppliersController : Controller {
    private ISupplierRepository supplierRepository;

    public SuppliersController(ISupplierRepository supplierRepo) {
      supplierRepository = supplierRepo;
    }

    public IActionResult Index() {
      ViewBag.SupplierEditId = TempData["SupplierEditId"];
      ViewBag.SupplierCreateId = TempData["SupplierCreateId"];
      return View(supplierRepository.GetAll());
    }

    public IActionResult Edit(long id) {
      TempData["SupplierEditId"] = id;
      return RedirectToAction(nameof(Index));
    }
```

```
    [HttpPost]
    public IActionResult Update(Supplier supplier) {
      supplierRepository.Update(supplier);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult Create(long id) {
      TempData["SupplierCreateId"] = id;
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Метод `Create()` устанавливает свойство `ViewBag`, которое применяется для отображения представления `Create.cshtml`, и затем перенаправляет браузер. Обратите внимание, что добавлять метод действия для операции создания необязательно. С точки зрения Entity Framework Core эта операция является обновлением существующего объекта `Supplier`, которое выполняется посредством метода `Update()`.

Чтобы увидеть эффект, перезапустите приложение, перейдите по ссылке `http://localhost:5000/suppliers` и щелкните на кнопке Add для одного из поставщиков. Заполните форму и щелкните на кнопке Save; вы заметите, что был создан новый объект `Product` (рис. 15.10).

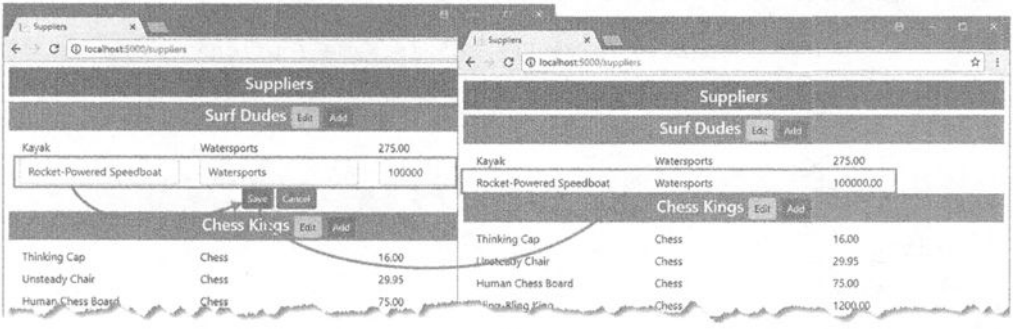

Рис. 15.10. Создание новых связанных данных

Изменение отношений

Операции создания и обновления упрощаются благодаря тому, что объекты `Product` связаны с объектом `Supplier`. Чуть больше работы требуется, когда дело доходит до изменения объекта `Supplier`, с которым связан объект `Product`, особенно для обязательного отношения, поскольку сервер баз данных не будет выполнять обновления, нарушающие целостность БД.

Для поддержки возможности изменения отношения добавьте в папку `Views/Suppliers` представление по имени `RelationshipEditor.cshtml` и поместите в него содержимое листинга 15.29.

Листинг 15.29. Содержимое файла `RelationshipEditor.cshtml` из папки `Views/Suppliers`

```
@model ValueTuple<Supplier, IEnumerable<Supplier>>
@{
  int counter = 0;
}
<form asp-action="Change" method="post">
  <input type="hidden" name="Id" value="@Model.Item1.Id" />
  <input type="hidden" name="Name" value="@Model.Item1.Name" />
  <input type="hidden" name="City" value="@Model.Item1.City" />
  <input type="hidden" name="State" value="@Model.Item1.State" />
  @foreach (Product p in Model.Item1.Products) {
    <input type="hidden" name="Products[@counter].Id" value="@p.Id" />
    <input type="hidden" name="Products[@counter].Name" value="@p.Name" />
    <input type="hidden" name="Products[@counter].Category"
          value="@p.Category" />
    <input type="hidden" name="Products[@counter].Price" value="@p.Price" />
    <div class="row">
      <div class="col">@p.Name</div>
      <div class="col">@p.Category</div>
      <div class="col">
        <select name="Products[@counter].SupplierId">
          @foreach (Supplier s in Model.Item2) {
            if (p.Supplier == s) {
              <option selected value="@s.Id">@s.Name</option>
            } else {
              <option value="@s.Id">@s.Name</option>
            }
          }
        </select>
      </div>
    </div>
    counter++;
  }
  <div class="row">
    <div class="col text-center m-1">
      <button class="btn btn-sm btn-danger" type="submit">Save</button>
      <a class="btn btn-sm btn-secondary" asp-action="Index">Cancel</a>
    </div>
  </div>
</form>
```

Моделью для представления является кортеж, содержащий объект `Supplier` и перечисление объектов `Supplier`. Это неуклюжий подход, но вы увидите, что весь процесс может оказаться неудобным, а получение двух объектов данных в качестве модели представления облегчает генерацию HTML-элементов, которые позволяют пользователю вносить изменения. Представление создает форму со скрытыми элементами, содержащими значения объектов `Supplier` и `Product`, которые не изменились, наряду с элементом `select`, позволяющим выбирать поставщика для каждого товара. Форма нацелена на действие по имени `Change`, которое вскоре будет определено.

Чтобы включить новое частичное представление в состав приложения, добавьте в представление `Index.cshtml` из папки `Views/Suppliers` элементы, показанные в листинге 15.30.

Листинг 15.30. Встраивание частичного приложения в файле `Index.cshtml` из папки `Views/Suppliers`

```
@model IEnumerable<DataApp.Models.Supplier>
@{
  ViewData["Title"] = "Suppliers";
  Layout = "_Layout";
}
@foreach (Supplier s in Model) {
  <h4 class="bg-info text-center text-white p-1">
    @s.Name
    <a asp-action="Edit" asp-route-id="@s.Id"
       class="btn btn-sm btn-warning">Edit</a>
    <a asp-action="Create" asp-route-id="@s.Id"
       class="btn btn-sm btn-danger">Add</a>
    <a asp-action="Change" asp-route-id="@s.Id"
       class="btn btn-sm btn-primary">Change</a>
  </h4>
  <div class="container-fluid">
    @if (s.Products == null || s.Products.Count() == 0) {
      <div class="p-1 text-center">No Products</div>
    } else if (ViewBag.SupplierEditId == s.Id) {
      @Html.Partial("Editor", s);
    } else if (ViewBag.SupplierRelationshipId == s.Id) {
      @Html.Partial("RelationshipEditor", (s, Model));
    } else {
      @foreach (Product p in s.Products) {
        <div class="row p-1">
          <div class="col">@p.Name</div>
          <div class="col">@p.Category</div>
          <div class="col">@p.Price</div>
        </div>
      }
      if (ViewBag.SupplierCreateId == s.Id) {
        @Html.Partial("Create", s);
      }
    }
  </div>
}
```

Новые элементы добавляют кнопку Change (Изменить), нацеленную на действие `Change` контроллера, которое установит свойство `ViewBag` по имени `SupplierRelationshipId`, используемое при принятии решения, когда показывать частичное представление из листинга 15.29. Чтобы добавить действия, на которые полагаются представления, определите в классе `SuppliersController` методы, как продемонстрировано в листинге 15.31.

Листинг 15.31. Добавление действий в файле SuppliersController.cs из папки Controllers

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
namespace DataApp.Controllers {
  public class SuppliersController : Controller {
    private ISupplierRepository supplierRepository;
    public SuppliersController(ISupplierRepository supplierRepo) {
      supplierRepository = supplierRepo;
    }
    public IActionResult Index() {
      ViewBag.SupplierEditId = TempData["SupplierEditId"];
      ViewBag.SupplierCreateId = TempData["SupplierCreateId"];
      ViewBag.SupplierRelationshipId = TempData["SupplierRelationshipId"];
      return View(supplierRepository.GetAll());
    }
    public IActionResult Edit(long id) {
      TempData["SupplierEditId"] = id;
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Update(Supplier supplier) {
      supplierRepository.Update(supplier);
      return RedirectToAction(nameof(Index));
    }
    public IActionResult Create(long id) {
      TempData["SupplierCreateId"] = id;
      return RedirectToAction(nameof(Index));
    }
    public IActionResult Change(long id) {
      TempData["SupplierRelationshipId"] = id;
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Change(Supplier supplier) {
      IEnumerable<Product> changed = supplier.Products
        .Where(p => p.SupplierId != supplier.Id);
      if (changed.Count() > 0) {
        IEnumerable<Supplier> allSuppliers
          = supplierRepository.GetAll().ToArray();
        Supplier currentSupplier
          = allSuppliers.First(s => s.Id == supplier.Id);
        foreach(Product p in changed) {
          Supplier newSupplier
            = allSuppliers.First(s => s.Id == p.SupplierId);
          newSupplier.Products = newSupplier.Products
            .Append(currentSupplier.Products
              .First(op => op.Id == p.Id)).ToArray();
          supplierRepository.Update(newSupplier);
        }
      }
```

```
            return RedirectToAction(nameof(Index));
        }
    }
}
```

Из-за конфликта между функциональными средствами код в методе `Change()`, принимающем запросы `POST`, получается довольно запутанным. Первым делом у БД запрашивается полный набор объектов `Supplier` и связанных с ними объектов `Product`, чтобы можно было обновить отношения, которые изменил пользователь. Инфраструктура Entity Framework Core отслеживает создаваемые ею объекты, а это значит, что выполнить обновление БД с применением объектов, созданных связывателем моделей MVC, невозможно без генерации исключения. Таким образом, объекты, созданные связывателем моделей MVC, придется обработать с целью выяснения, что должно изменяться, и затем перенести изменения на объекты, созданные инфраструктурой Entity Framework Core, которые могут использоваться для обновления БД. (Не тратьте слишком много времени на исследование кода в листинге 15.31, потому что в последующих разделах будут показаны более простые подходы.)

Чтобы увидеть, как можно изменять отношения, перезапустите приложение, перейдите по ссылке `http://localhost:5000/suppliers` и щелкните на одной из кнопок Change. С помощью раскрывающихся списков измените отношения и щелкните на кнопке Save для обновления БД. Объекты `Product`, чьи отношения были отредактированы, отобразятся вместе с их новыми объектами `Supplier` (рис. 15.11).

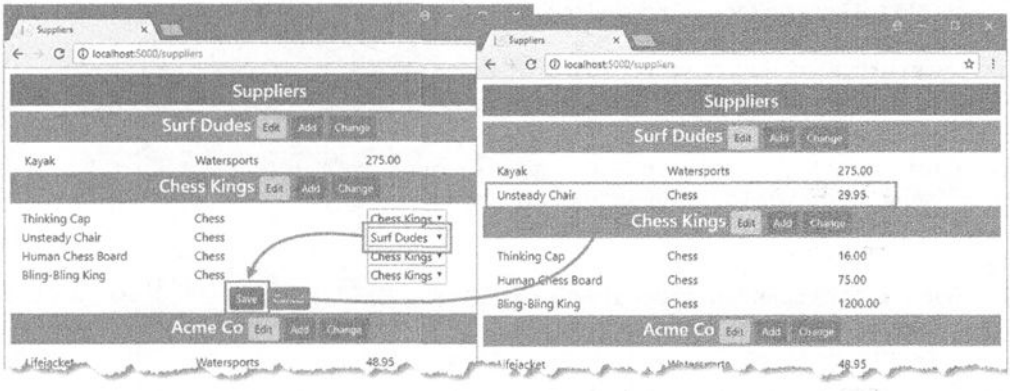

Рис. 15.11. Изменение отношений между объектами

Упрощение кода изменения отношений

Код, предназначенный для обработки обновлений в листинге 15.31, полагается на последовательность запросов LINQ, которые извлекают данные из БД и объединяют изменения из объектов, созданных на основе HTTP-запроса. Такой подход обходит систему отслеживания объектов Entity Framework Core, поддерживающую средства вроде отслеживания изменений и исправления. Эти средства полезны, но система отслеживания изменений мешает выполнению других операций.

Код обработки изменений можно упростить за счет отключения средства отслеживания объектов, как демонстрируется в листинге 15.32, что позволит применять объекты, созданные связывателем моделей MVC, для обновления БД.

Листинг 15.32. Отключение средства отслеживания объектов в файле SuppliersController.cs из папки Models

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;

namespace DataApp.Controllers {

  public class SuppliersController : Controller {
    private ISupplierRepository supplierRepository;
    private EFDatabaseContext dbContext;

    public SuppliersController(ISupplierRepository supplierRepo,
        EFDatabaseContext context) {
      supplierRepository = supplierRepo;
      dbContext = context;
    }

    public IActionResult Index() {
      ViewBag.SupplierEditId = TempData["SupplierEditId"];
      ViewBag.SupplierCreateId = TempData["SupplierCreateId"];
      ViewBag.SupplierRelationshipId = TempData["SupplierRelationshipId"];
      return View(supplierRepository.GetAll());
    }

    public IActionResult Edit(long id) {
      TempData["SupplierEditId"] = id;
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Update(Supplier supplier) {
      supplierRepository.Update(supplier);
      return RedirectToAction(nameof(Index));
    }

    public IActionResult Create(long id) {
      TempData["SupplierCreateId"] = id;
      return RedirectToAction(nameof(Index));
    }

    public IActionResult Change(long id) {
      TempData["SupplierRelationshipId"] = id;
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Change(Supplier supplier) {
      IEnumerable<Product> changed
        = supplier.Products.Where(p => p.SupplierId != supplier.Id);
      IEnumerable<long> targetSupplierIds
        = changed.Select(p => p.SupplierId).Distinct();
      if (changed.Count() > 0) {
        IEnumerable<Supplier> targetSuppliers = dbContext.Suppliers
```

```
            .Where(s => targetSupplierIds.Contains(s.Id))
            .AsNoTracking().ToArray();
          foreach(Product p in changed) {
            Supplier newSupplier
              = targetSuppliers.First(s => s.Id == p.SupplierId);
            newSupplier.Products = newSupplier.Products == null
              ? new Product[] { p }
                : newSupplier.Products.Append(p).ToArray();
          }
          dbContext.Suppliers.UpdateRange(targetSuppliers);
          dbContext.SaveChanges();
        }
        return RedirectToAction(nameof(Index));
      }
    }
}
```

Ради простоты класс контекста БД используется напрямую вместо обработки изменений через интерфейс хранилища и класс его реализации. Код в листинге 15.33 проще (хотя может и не выглядеть таковым), поскольку средство отслеживания было отключено с применением расширяющего метода `AsNoTracking()` в запросе, который извлекает из БД объект `Supplier` и связанные объекты `Product`:

```
...
IEnumerable<Supplier> targetSuppliers = dbContext.Suppliers
  .Where(s => targetSupplierIds.Contains(s.Id))
  .AsNoTracking().ToArray();
...
```

Когда используется метод `AsNoTracking()`, инфраструктура Entity Framework Core не отслеживает создаваемые ею объекты, что позволяет применять объекты `Product`, созданные связывателем моделей MVC, для обновления БД.

Код в листинге 15.32 проще кода в листинге 15.31 из-за того, что для выполнения обновлений не приходится извлекать объекты, связанные с каждым объектом `Supplier`. Инфраструктура Entity Framework Core не обновляет данные, которые не были запрошены из БД, а потому исключение объектов `Product` из запроса означает, что существующие отношения не затрагиваются изменениями, указанными пользователем.

Внимание! Метод `AsNoTracking()` должен использоваться с осторожностью, поскольку он препятствует работе других полезных средств, таких как обнаружение изменений и исправление.

Дальнейшее упрощение кода изменения отношений

В предыдущих двух листингах демонстрировалась возможность изменения отношений на стороне "один" отношения "один ко многим", но поступать подобным образом неудобно. Существует гораздо более простой способ решения такой задачи, который можно понять, подумав о том, как отношение представляется в БД.

В случае примера приложения класс `Product` имеет свойство `SupplierId`, которое хранит значение свойства `Id` для связанного объекта `Supplier`. Свойство

Products, определенное в классе Supplier, предназначено только для удобства навигации, и когда отношение для объекта Product изменяется, инфраструктуре Entity Framework Core приходится только обновить строку в таблице Products, чтобы отразить изменение, даже если оно вносится через навигационное свойство IEnumerable<Product>.

Осознав, как производится обновление, можно значительно упростить код для выполнения обновлений за счет оперирования объектами Product напрямую (листинг 15.33).

Листинг 15.33. Выполнение прямых обновлений в файле SuppliersController.cs из папки Controllers

```
...
[HttpPost]
public IActionResult Change(long Id, Product[] products) {
  dbContext.Products.UpdateRange(products.Where(p => p.SupplierId != Id));
  dbContext.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Изменение параметров для метода действия сообщает связывателю моделей MVC о том, что требуется значение Id для объекта Supplier, отношения которого пользователь изменяет, и набор объектов Product, связанных с этим объектом Supplier.

Чтобы отфильтровать объекты Product, которые не изменились, применяется LINQ. Изменившиеся объекты Product передаются методу DbSet<T>.UpdateRange(), предоставляемому классом контекста БД, который позволяет обновлять множество объектов за раз. Затем вызывается метод SaveChanges() для отправки изменений БД и браузер перенаправляется на действие Index.

Результат будет таким же, как у кода из листингов 15.31 и 15.32, но код в листинге 15.33 проще и легче для понимания. Приведенный пример показал, что выполнять обновления допускается с использованием любого навигационного свойства в отношении "один ко многим". Но размышления о том, как обновления будут отражаться в БД, может дать представление о том, удастся ли получить более простой результат, отдавая предпочтение одному свойству перед другим.

Резюме

В главе был продемонстрирован способ доступа к связанным данным напрямую либо путем повышения данных до свойства контекста, либо за счет применения методов контекста, которые принимают параметр типа. После повышения данных часто возникает необходимость перехода от них на другие части модели данных, а потому было показано, как укомплектовывать отношение добавлением навигационного свойства, чтобы определить отношение "один ко многим", которое чаще всего встречается при разработке с помощью ASP.NET Core MVC и Entity Framework Core. Рассматривались различные способы запрашивания связанных данных в отношении "один ко многим", а также создания и обновления связанных данных через навигационное свойство этого отношения. В заключение была продемонстрирована возможность редактирования отношений через навигационное свойство, но более простые результаты можно получить, если принять во внимание то, каким образом отношения представлены в БД. В следующей главе будет продолжено обсуждение средств Entity Framework Core, предназначенных для использования отношений.

ГЛАВА 16

Работа с отношениями, часть 2

Отношение "один ко многим", которое было описано в главе 15, не является единственным видом отношений, поддерживаемых инфраструктурой Entity Framework Core. В этой главе будет показано, как определять отношения *"один к одному"* и *"многие ко многим"*, продемонстрировано, каким образом запрашивать связанные данные при использовании таких отношений, и объяснено, как ими управлять в БД. В табл. 16.1 приведена сводка по главе.

Таблица 16.1. Сводка по главе

Задача	Решение	Листинг
Определение укомплектованного отношения "один к одному"	Добавьте обоюдные навигационные свойства и свойство внешнего ключа в зависимый сущностный класс	16.1–16.7
Обновление связанных объектов	Работайте со связанными объектами отдельно, чтобы гарантировать корректность добавления и обновления данных	16.8–16.16
Определение отношения "многие ко многим"	Создайте соединяющий класс, который имеет отношения "один ко многим" с двумя связанными типами данных	16.17–16.28

Подготовительные шаги

В главе продолжается работа с проектом `DataApp`, созданным в главе 11 и модифицированным в последующих главах. В качестве подготовки откройте окно командной строки, перейдите в папку `DataApp` и выполните команды из листинга 16.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 16.1. Переустановка БД

```
dotnet ef database drop --force --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Команды удаляют и воссоздают БД, применяемую для хранения объектов `Product`, а также их связанных объектов `Supplier`, `ContactDetails` и `ContactLocation`, что поможет обеспечить получение ожидаемых результатов в примерах, приводимых в главе.

Запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`. Приложение заполнит БД начальными данными и отобразит список товаров (рис. 16.1).

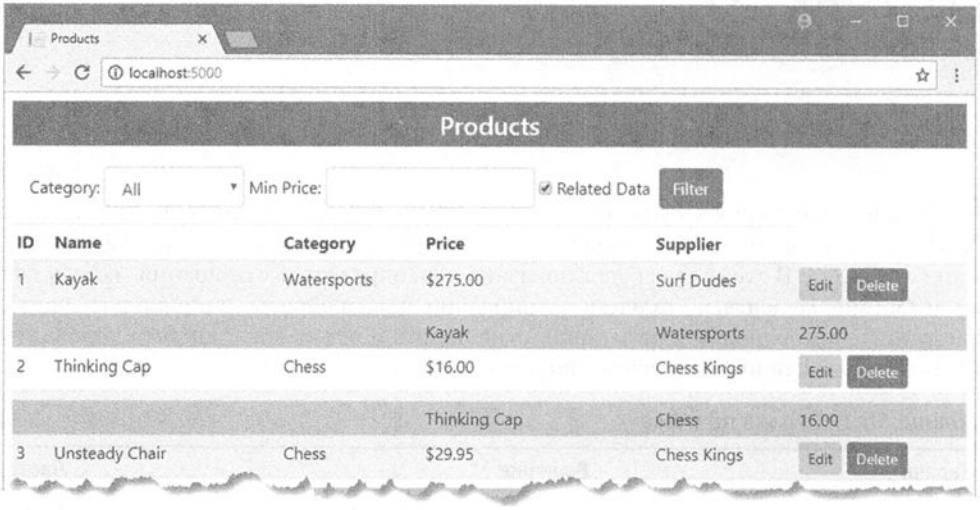

Рис. 16.1. Выполнение примера приложения

Укомплектование отношения "один к одному"

В отношении "один к одному" объект одного типа связан с единственным объектом другого типа. Для завершения отношения "один к одному" требуются два шага, которые объясняются в последующих разделах на примере отношения между классами `Supplier` и `ContactDetails`, обеспечивающего связь каждого объекта `Supplier` с одним объектом `ContactDetails`.

Определение навигационного свойства

Первый шаг заключается в определении навигационного свойства. Оно является противоположностью свойства, которое определялось в главе 14 для создания отношения между классами `Supplier` и `ContactDetails`. Когда определяется только одно навигационное свойство, инфраструктура Entity Framework Core предполагает, что требуется отношение "один ко многим", и создать отношение "один к одному" можно только в случае, если определены оба навигационных свойства. Добавьте в класс `ContactDetails` навигационное свойство, которое укомплектует отношение, как показано в листинге 16.2.

Листинг 16.2. Добавление свойства в файле `ContactDetails.cs` из папки `Models`

```
namespace DataApp.Models {

  public class ContactDetails {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Phone { get; set; }
    public ContactLocation Location { get; set; }

    public Supplier Supplier { get; set; }
  }
}
```

Свойство `Supplier` возвращает одиночный объект `Supplier`, а не перечисление `IEnumerable<T>`, возвращаемое навигационным свойством для отношения "один ко многим". Результат навигационного свойства в форме одиночного объекта сообщает инфраструктуре Entity Framework Core о том, что создается отношение "один к одному". При установлении типа отношения имя навигационного свойства во внимание не принимается.

Выбор зависимого сущностного класса

Второй шаг предусматривает принятие решения о том, какой класс в отношении будет *зависимой сущностью*, и определение внутри него свойства внешнего ключа. Инфраструктура Entity Framework Core хранит объекты в виде строк внутри таблиц БД. Когда создается отношение между классами, Entity Framework Core добавляет в одну из таблиц столбец и применяет его для записи значения первичного ключа связанного объекта. Класс, чья таблица содержит столбец внешнего ключа, называется *зависимой сущностью*, а другой класс (таблица которого не содержит столбец внешнего ключа) — *главной сущностью*.

В отношении "один ко многим" зависимой сущностью всегда является класс на стороне "многие", поэтому инфраструктуре Entity Framework Core известно, где должен определяться столбец внешнего ключа. Но в отношении "один к одному" зависимой сущностью может быть любой из двух классов, и только свойство внешнего ключа предоставляет инфраструктуре Entity Framework Core информацию, необходимую для помещения столбца внешнего ключа в правильное место БД.

В рассматриваемом примере зависимой сущностью будет класс `ContactDetails`, а потому определите свойство внешнего ключа в соответствии с соглашениями, описанными в главе 14, как показано в листинге 16.3.

Выбор зависимого сущностного класса

При создании отношения между классами иногда трудно понять, где должно быть определено свойство внешнего ключа. Нужно задать себе вопрос: способен ли объект типа `X` существовать без связанного объекта типа `Y`? Если ответ положительный, тогда тип `X` — главная сущность и свойство внешнего ключа должно определяться в классе `Y`, который является зависимой сущностью. Если же ответ отрицательный, то главной сущностью будет тип `Y` и свойство внешнего ключа должно определяться в классе `X` — зависимой сущности. Например, в случае приложения `DataApp` вопрос выглядит так: способен ли объект типа `Supplier` существовать без связанного объекта типа `ContactDetails`?

Временами ответ на указанный вопрос очевиден, что облегчает определение, куда помещать свойство внешнего ключа. В приложении DataApp существование поставщика товара без контакта имеет смысл, но обратное бессмысленно. Именно потому класс ContactDetails сделан зависимой сущностью.

Однако зачастую ответ не вполне ясен, особенно когда дело касается абстрактных концепций, не имеющих аналогов в реальном мире. Для таких отношений лучшее, что можно предпринять — сделать обоснованное предположение и посмотреть, какую форму примет приложение. Подход далек от идеала, но работать с данными не всегда легко, а ошибку можно исправить за счет перемещения свойства внешнего ключа с последующим созданием и применением новой миграции.

Листинг 16.3. Определение свойства внешнего ключа в файле ContactDetails.cs из папки Models

```
namespace DataApp.Models {
  public class ContactDetails {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Phone { get; set; }
    public ContactLocation Location { get; set; }
    public long SupplierId { get; set; }
    public Supplier Supplier { get; set; }
  }
}
```

Тип свойства внешнего ключа устанавливает, каким будет отношение — обязательным или необязательным. В листинге 16.3 для свойства указан тип long, что приведет к созданию обязательного отношения, т.е. объект ContactDetails не может быть сохранен в БД, если он не связан с объектом Supplier. В случае использования типа, который допускает значения null, наподобие long?, было бы создано необязательное отношение.

Внимание! Всегда укомплектовывайте отношение посредством свойства внешнего ключа. Если свойство внешнего ключа в отношении "один к одному" не указано, тогда инфраструктура Entity Framework Core попытается вывести класс, являющийся зависимой сущностью, что может привести к непредсказуемым результатам.

Создание и применение миграции

После определения навигационного свойства и свойства внешнего ключа можно создать миграцию. Выполните команду из листинга 16.4, чтобы создать миграцию, которая изменит отношение между объектами ContactDetails и Supplier.

Листинг 16.4. Создание миграции для отношения "один к одному"

```
dotnet ef migrations add CompleteOneToOne --context EFDatabaseContext
```

Чтобы понять, каким образом завершение отношения изменит БД, откройте файл <отметка времени>_CompleteOneToOne.cs, созданный в папке Migrations, и просмотрите метод Up().

Когда было только навигационное свойство, инфраструктура Entity Framework Core создавала отношение "один ко многим", в котором класс Supplier являлся зависимой сущностью. Для отражения нового отношения миграция удаляет существующий столбец внешнего ключа и создает в таблице ContactDetails новый такой столбец, что делает класс ContactDetails зависимой сущностью. Тот факт, что это отношение "один к одному", отражается включением в миграцию следующего оператора:

```
...
migrationBuilder.CreateIndex(
  name: "IX_ContactDetails_SupplierId", table: "ContactDetails",
  column: "SupplierId", unique: true);
...
```

Отношение "один к одному" представляется созданием индекса, который требует уникальных значений в столбце внешнего ключа, что гарантирует возможность связывания объекта ContactDetails только с одним объектом Supplier.

Совет. Миграция не может быть применена к БД, поскольку существующие данные конфликтуют с новыми ограничениями, налагаемыми миграцией. По этой причине БД будет удалена и воссоздана, так что при запуске приложения она заполнится начальными данными.

Работа с отношениями "один к одному"

После того, как отношение укомплектовано, выполнение операций над связанными данными становится прямолинейным, как описано в последующих разделах.

Запрашивание связанных данных в отношении "один к одному"

Запрашивание связанных данных в отношении "один к одному" — простая задача, потому что вам известно, что имеется только один связанный объект, с которым приходится иметь дело. Метод Include() используется для сообщения инфраструктуре Entity Framework Core о следовании по навигационному свойству и запрашивании связанного объекта в отношении "один к одному". Вы можете начинать запрос с объектов на любой стороне отношения, и в главе будут запрашиваться объекты ContactDetails с переходом на связанные объекты Supplier.

Чтобы продемонстрировать работу с отношением "один к одному", добавьте в папку Controllers файл класса по имени One2OneController.cs с определением контроллера, приведенным в листинге 16.5.

На заметку! Как объяснялось в главе 11, в реальных проектах рекомендуется применять хранилище, но в этом контроллере работа производится напрямую с классом контекста БД во избежание внесения трех изменений (в контроллер, интерфейс хранилища и класс реализации хранилища) каждый раз, когда иллюстрируется очередное средство или операция.

Листинг 16.5. Содержимое файла One2OneController.cs из папки Controllers

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace DataApp.Controllers {

  public class One2OneController : Controller {
    private EFDatabaseContext context;
    public One2OneController(EFDatabaseContext ctx) => context = ctx;
    public IActionResult Index () {
      return View(context.Set<ContactDetails>().Include(cd => cd.Supplier));
    }
  }
}
```

В контроллере определен метод действия по имени `Index()`, который запрашивает у БД все объекты `ContactDetails` вместе со связанными объектами и использует их в качестве модели представления для стандартного представления. Чтобы отобразить данные пользователю, создайте папку `Views/One2One` и добавьте в нее файл представления по имени `Index.cshtml` с содержимым из листинга 16.6.

Листинг 16.6. Содержимое файла Index.cshtml из папки Views/One2One

```
@model IEnumerable<DataApp.Models.ContactDetails>
@{
  ViewData["Title"] = "ContactDetails";
  Layout = "_Layout";
}

<table class="table table-striped table-sm">
  <tr>
    <th>ID</th>
    <th>Name</th>
    <th>Phone</th>
    <th></th>
    <th class="table-dark text-center" colspan="4">Supplier</th>
  </tr>
  @foreach (var s in Model) {
    <tr>
      <td>@s.Id</td>
      <td>@s.Name</td>
      <td>@s.Phone</td>
      <td>
        <form>
          <button class="btn btn-sm btn-warning"
                asp-action="Edit" asp-route-id="@s.Id">
            Edit
          </button>
        </form>
      </td>
      @if (s.Supplier != null) {
        <td class="table-dark">@s.Supplier.Id</td>
        <td class="table-dark">@s.Supplier.Name</td>
```

```
        <td class="table-dark">@s.Supplier.City</td>
        <td class="table-dark">@s.Supplier.State</td>
      } else {
        <td colspan="4" class="table-dark text-center">
          No Related Supplier
        </td>
      }
    </tr>
  }
</table>
<a class="btn btn-primary" asp-action="Create">Create</a>
```

Представление создает таблицу, в каждой строке которой отображается объект `ContactDetails` и связанный объект `Supplier`. Имеются также кнопка и якорный элемент, которые нацелены на методы действий, пока еще отсутствующие в контроллере, но которые будут применяться для демонстрации разнообразных средств в последующих разделах.

Обновление базы данных и выполнение приложения

Если вы попытаетесь применить миграцию, созданную в БД, то получите ошибку, поскольку существующие данные конфликтуют с изменениями, которые содержит миграция. В случае производственной БД вам пришлось бы уделить время на перенос данных, но для процесса разработки БД можно удалить и воссоздать. Выполните в папке проекта `DataApp` команды из листинга 16.7, чтобы удалить БД и применить миграцию для отношения "один к одному".

Листинг 16.7. Удаление и воссоздание БД

```
dotnet ef database drop --force --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000/one2one`. При запуске приложения БД заполнится начальными данными, а запрошенный URL будет нацелен на метод действия `Index()`, определенный классом контроллера в листинге 16.5, который выбирает представление, созданное в листинге 16.6, и отображает результаты, как показано на рис. 16.2.

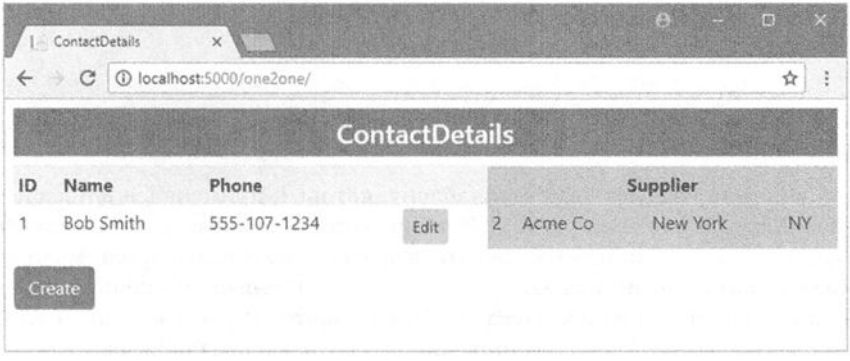

Рис. 16.2. Отображение связанных данных в отношении "один к одному"

Создание и обновление связанных объектов

Связанные объекты можно создавать и обновлять через любое из двух навигационных свойств в отношении "один к одному". Это означает, например, что вы можете оперировать с объектом `Supplier` через свойство `Supplier` объекта `ContactDetails` или в равной степени оперировать с объектом `ContactDetails` через свойство `Contact` объекта `Supplier`. Внимание в настоящем разделе будет сосредоточено на данных `ContactDetails`, так что добавьте в контроллер `One2One` методы действий, выделенные полужирным в листинге 16.8, для поддержки процессов редактирования и создания.

Листинг 16.8. Добавление методов действий в файле `One2OneController.cs` из папки `Controllers`

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace DataApp.Controllers {

  public class One2OneController : Controller {
    private EFDatabaseContext context;

    public One2OneController(EFDatabaseContext ctx) => context = ctx;
    public IActionResult Index () {
      return View(context.Set<ContactDetails>().Include(cd => cd.Supplier));
    }
    public IActionResult Create() => View("ContactEditor");
    public IActionResult Edit(long id) {
      return View("ContactEditor",
        context.Set<ContactDetails>()
          .Include(cd => cd.Supplier).First(cd => cd.Id == id));
    }
    [HttpPost]
    public IActionResult Update(ContactDetails details) {
      if (details.Id == 0) {
        context.Add<ContactDetails>(details);
      } else {
        context.Update<ContactDetails>(details);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Новые действия следуют тому же шаблону, который применялся в предшествующих главах. Методы `Create()` и `Edit()` используются для выбора представления по имени `ContactEditor`, которое позволит пользователю создавать или модифицировать объект с помощью метода `EditContact()`, применяющего хранилище для запрашивания выбранного пользователем объекта. Метод `Update()` используется для получения запросов `POST`, содержащих данные, которые ввел пользователь, и вызывает метод `Add()` или `Update()` объекта контекста для обновления БД.

Чтобы предоставить представление, которое даст пользователю возможность создавать или редактировать объект, добавьте в папку `Views/One2One` файл представления по имени `ContactEditor.cshtml` с элементами, приведенными в листинге 16.9.

Листинг 16.9. Содержимое файла `ContactEditor.cshtml` из папки `Views/One2One`

```
@model DataApp.Models.ContactDetails
@{
  ViewData["Title"] = Model == null ? "Create" : "Edit";
  Layout = "_Layout";
}
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="Supplier.Id" />
  <h4>Contact Details</h4>
  <div class="p-1 m-1">
    <div class="form-row">
      <div class="form-group col">
        <label asp-for="Name" class="form-control-label"></label>
        <input asp-for="Name" class="form-control" />
      </div>
      <div class="form-group col">
        <label asp-for="Phone" class="form-control-label"></label>
        <input asp-for="Phone" class="form-control" />
      </div>
    </div>
  </div>
  <h4>Supplier</h4>
  <div class="p-1 m-1">
    <div class="form-row">
      <div class="form-group col">
        <label asp-for="Supplier.Name" class="form-control-label"></label>
        <input asp-for="Supplier.Name" class="form-control" />
      </div>
      <div class="form-group col">
        <label asp-for="Supplier.City" class="form-control-label"></label>
        <input asp-for="Supplier.City" class="form-control" />
      </div>
      <div class="form-group col">
        <label asp-for="Supplier.State" class="form-control-label"></label>
        <input asp-for="Supplier.State" class="form-control" />
      </div>
    </div>
    @if (ViewBag.Suppliers != null) {
      @Html.Partial("RelationshipEditor", Model.SupplierId)
    }
  </div>
  <div class="text-center m-1">
    <button type="submit" class="btn btn-primary">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Самыми важными элементами в этом представлении являются те, которые пользователь не увидит: элементы `hidden`, содержащие значения свойств `Id` объектов `ContactDetails` и `Supplier`. Если значения свойств `Id` нулевые, тогда будут созданы новые объекты; для любого другого значения будут обновлены существующие объекты. (Представление `ContactEditor` также ссылается на частичное представление, которое создается в следующем разделе.)

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000/one2one` и щелкните на кнопке Create (Создать). Заполните элементы формы и щелкните на кнопке Save (Сохранить), чтобы создать новые объекты `ContactDetails` и `Supplier`, которые затем отобразятся (рис. 16.3).

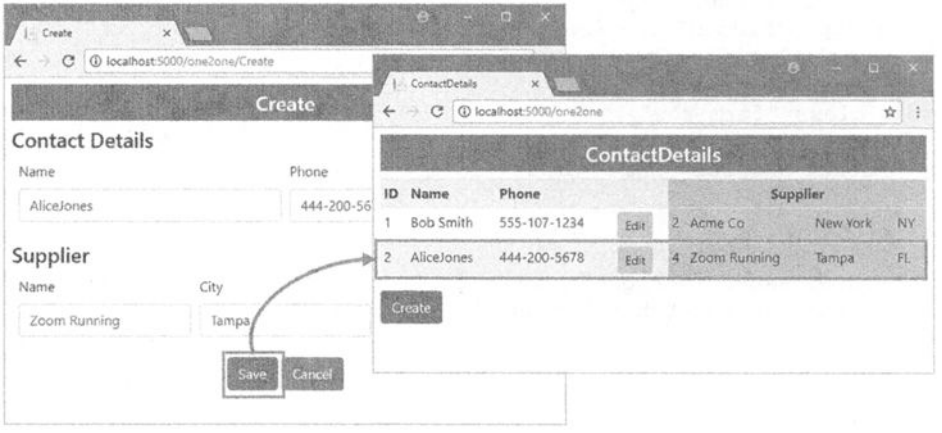

Рис. 16.3. Создание новых объектов через навигационное свойство

Щелкните на кнопке Edit (Редактировать) для вновь созданного объекта `Supplier` и измените значение в поле Name (Название). После щелчка на кнопке Save инфраструктура Entity Framework Core проследует по навигационному свойству объекта `ContactDetails`, созданному связывателем моделей MVC, и обновит БД (рис. 16.4).

Рис. 16.4. Обновление существующего объекта через навигационное свойство

Изменение отношения "один к одному"

При изменении отношений между объектами должна соблюдаться осторожность, особенно в случае обязательных отношений. Инфраструктура Entity Framework Core не навязывает ограничения перед отправкой обновлений БД, поэтому легко выполнить обновление, которое нарушит ссылочную целостность БД, приводя к ошибке. В последующих разделах будет показано, как обновлять обязательные и необязательные отношения, начиная с существующего обязательного отношения, которое уже сконфигурировано в БД для примера приложения.

Изменение обязательного отношения "один к одному"

Сложность с обязательным отношением заключается в том, что необходимо избегать сохранения любой зависимой сущности, которая не связана с главной сущностью. В примере приложения это означает, что каждый объект `ContactDetails` должен быть ассоциирован с объектом `Supplier`. Попытка сохранить или обновить объект `ContactDetails`, который не связан с каким-либо объектом `Supplier`, приведет к ошибке.

Обязательное отношение применяется только в одном направлении; зависимая сущность должна быть связана с главной сущностью, но главная сущность не обязана быть связанной с зависимой сущностью. В примере приложения сказанное означает, что объекты `Supplier` могут существовать, не будучи связанными с объектами `ContactDetails`. Это соответствует тому, как отношения представлены в БД, где в таблице, используемой для хранения зависимых сущностей, определяется столбец внешнего ключа, но в таблице, предназначенной для хранения главных сущностей, вообще не содержится какой-либо информации об отношении.

В итоге возникают два сценария с изменением отношения объекта. Первый сценарий касается ситуации, когда нужно изменить отношение так, чтобы зависимая сущность была связана с главной сущностью, которая в текущий момент не находится в отношении. В примере приложения это означает наличие несвязанных (или "запасных") объектов `Supplier`, один из которых станет связанным с объектом `ContactDetails`, заменяя существующий объект `Supplier`. В конце операции объект `Supplier`, с которым был первоначально связан объект `ContactDetails`, становится одним из "запасных".

Второй сценарий представляет ситуацию, когда требуется создать отношение с объектом `Supplier`, который уже связан с другим объектом. Существующий объект `ContactDetails` можно оставить несвязанным, не нарушая ограничения БД, так что необходимо создать отношение с другим объектом `Supplier` обычно за счет выполнения обмена местами.

Чтобы добавить поддержку для изменения существующего отношения, внесите в контроллер `One2One` изменения, выделенные полужирным в листинге 16.10.

Листинг 16.10. Изменение отношений в файле `One2OneController.cs` из папки `Controllers`

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace DataApp.Controllers {
```

```
public class One2OneController : Controller {
  private EFDatabaseContext context;
  public One2OneController(EFDatabaseContext ctx) => context = ctx;
  public IActionResult Index () {
    return View(context.Set<ContactDetails>().Include(cd => cd.Supplier));
  }
  public IActionResult Create() => View("ContactEditor");
  public IActionResult Edit(long id) {
    ViewBag.Suppliers = context.Suppliers.Include(s => s.Contact);
    return View("ContactEditor",
      context.Set<ContactDetails>()
        .Include(cd => cd.Supplier).First(cd => cd.Id == id));
  }
  [HttpPost]
  public IActionResult Update(ContactDetails details) {
    if (details.Id == 0) {
      context.Add<ContactDetails>(details);
    } else {
      context.Update<ContactDetails>(details);
    }
    context.SaveChanges();
    return RedirectToAction(nameof(Index));
  }
}
}
```

Средство строится пошагово, т.к. код может вызвать путаницу. Единственное изменение, внесенное в настоящий момент, касается создания свойства `ViewBag`, которое будет снабжать представление данными `Supplier`. В результате будет обеспечено включение в вывод частичного представления по имени `RelationshipEditor.cshtml`, когда пользователь редактирует объект, что дает возможность отобразить список существующих объектов `Supplier`, с которыми можно устанавливать отношение. Создайте в папке `Views/One2One` файл представления `RelationshipEditor.cshtml` с содержимым из листинга 16.11.

Листинг 16.11. Содержимое файла `RelationshipEditor.cshtml` из папки `Views/One2One`

```
@model long

<div class="p-1 m-1">
  @foreach (Supplier s in ViewBag.Suppliers) {
    @if (s.Id != Model) {
      <div class="form-row">
        <div class="form-group col">
          <input type="radio" name="targetSupplierId" value="@s.Id" />
          @if (s.Contact == null) {
            <input type="hidden" name="spares" value="@s.Id" />
          }
        </div>
        <div class="form-group col-1">
          <label class="form-control-label">@s.Id</label>
        </div>
```

```
        <div class="form-group col">
          <label class="form-control-label">@s.Name</label>
        </div>
        <div class="form-group col">
          <label class="form-control-label">@s.City</label>
        </div>
        <div class="form-group col">
          <label class="form-control-label">@s.State</label>
        </div>
        <div class="form-group col">
          <label class="form-control-label">
            @(s.Contact == null ? "(None)" : s.Contact.Name)
          </label>
        </div>
      </div>
    }
  }
</div>
```

Частичное представление `RelationshipEditor` перечисляет последовательность объектов `Supplier` для отображения пользователю списка, позволяющего выбрать объект `Supplier` с применением переключателя, который помещает значение данных формы по имени `targetSupplierId` в HTTP-запрос `POST`, отправляемый приложению. Также имеется коллекция значений `spares`, содержащая значения первичного ключа свободных объектов `Supplier`, которая будет использоваться для выяснения, существует ли подлежащее изменению отношение. Запустите приложение с помощью `dotnet run`, перейдите по ссылке `http://localhost:5000/one2one` и щелкните на кнопке Edit для одного из отображаемых объектов `ContactDetails`. В дополнение к элементам `input` из предшествующих примеров вы увидите список объектов `Supplier`, исключая тот, с которым связан текущий объект `ContactDetails` (рис. 16.5).

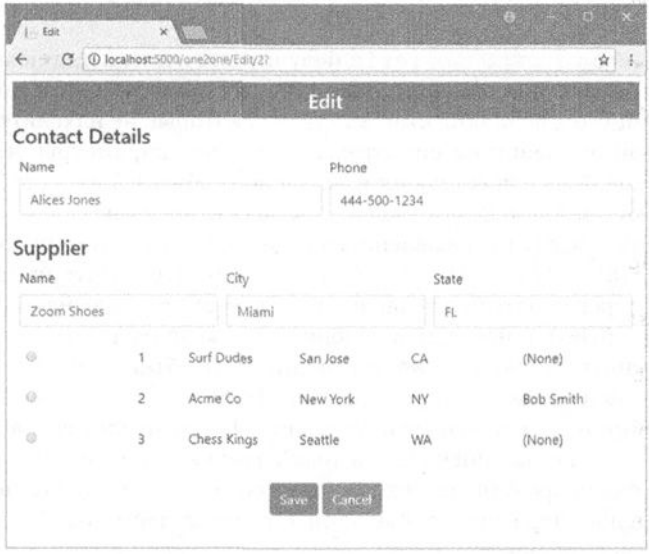

Рис. 16.5. Отображение списка поставщиков

После завершения всех подготовительных шагов можно добавить код, который будет изменять отношение. Первый сценарий описывает ситуацию, когда пользователь выбирает объект Supplier, в текущий момент не связанный с каким-либо объектом ContactDetails. Такое изменение сделать легко, поскольку понадобится лишь изменить значение свойства SupplierId объекта ContactDetails на первичный ключ "запасного" объекта Supplier. Изменение разорвет отношение с текущим объектом Supplier и создаст отношение с новым объектом Supplier. В листинге 16.12 показаны изменения, которые должны быть внесены в контроллер для поддержки этого сценария.

Листинг 16.12. Изменение отношений в файле One2OneController.cs из папки Controllers

```
...
[HttpPost]
public IActionResult Update(ContactDetails details,
    long? targetSupplierId, long[] spares) {
  if (details.Id == 0) {
    context.Add<ContactDetails>(details);
  } else {
    context.Update<ContactDetails>(details);
    if (targetSupplierId.HasValue) {
      if (spares.Contains(targetSupplierId.Value)) {
        details.SupplierId = targetSupplierId.Value;
      }
    }
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Новый параметр targetSupplierId получит значение, когда пользователь выберет один из существующих объектов Supplier, а параметр spares будет содержать значения первичного ключа объектов Supplier, которые не находятся в отношении. Добавленные в метод действия операторы проверяют параметры, чтобы выяснить, выбрал ли пользователь существующий "запасной" объект Supplier, и если это так, то устанавливают свойство SupplierId, создавая отношение.

Чтобы увидеть эффект от добавленного кода, перезапустите приложение и перейдите по ссылке http://localhost:5000/one2one. Щелкните на одной из кнопок Edit и выберите переключатель для какого-то объекта Supplier из числа тех, которые содержат (None) ((Нет)) в последней колонке, указывая на то, что они являются "запасными". Щелкните на кнопке Save; вы увидите, что объект ContactDetails теперь связан с выбранным объектом Supplier (рис. 16.6).

Второй сценарий может вызвать путаницу. Когда нужно создать отношение с объектом Supplier не из списка "запасных", требуются дополнительные шаги для обновления БД с одновременным сохранением ее ссылочной целостности. В листинге 16.13 приведены операторы, добавленные в метод действия Update(), которые поддерживают такой процесс.

Рис. 16.6. Выбор свободного поставщика

Листинг 16.13. Изменение существующих отношений в файле `One2OneController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(ContactDetails details,
    long? targetSupplierId, long[] spares) {
  if (details.Id == 0) {
    context.Add<ContactDetails>(details);
  } else {
    context.Update<ContactDetails>(details);
    if (targetSupplierId.HasValue) {
      if (spares.Contains(targetSupplierId.Value)) {
        details.SupplierId = targetSupplierId.Value;
      } else {
        ContactDetails targetDetails = context.Set<ContactDetails>()
          .FirstOrDefault(cd => cd.SupplierId == targetSupplierId);
        targetDetails.SupplierId = details.Supplier.Id;
        Supplier temp = new Supplier { Name = "temp" };
        details.Supplier = temp;
        context.SaveChanges();

        temp.Contact = null;
        details.SupplierId = targetSupplierId.Value;
        context.Suppliers.Remove(temp);
      }
    }
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Если выбранный пользователем объект Supplier уже связан с объектом ContactDetails, тогда придется сделать ряд перестановок. Прежде всего, у БД запрашивается объект ContactDetails, связанный с указанным пользователем объектом Supplier, и изменяется его свойство SupplierId, чтобы оно относилось к объекту, который пользователь редактирует. (В конце раздела будет рассмотрен специфический пример, который поможет понять, что происходит в описанной последовательности обновлений.)

В этот момент обновлять БД нельзя, т.к. два объекта ContactDetails связаны с одним и тем же объектом Supplier, что в БД не разрешено. И поскольку отношение "один ко многим" является обязательным, сохранить любой из двух объектов ContactDetails не удастся до тех пор, пока он не будет связан с объектом Supplier.

Таким образом, на следующем шаге создается *новый* объект Supplier, который присваивается переменной по имени temp и применяется для создания отношения с редактируемым объектом ContactDetail. В данный момент вызывается метод SaveChanges() для сохранения изменений, и в таблице Suppliers будет создана новая строка для объекта temp.

В результате обновления выбранный пользователем объект Supplier становится "запасным" и доступным для использования. Значение его первичного ключа присваивается свойству SupplierId редактируемого объекта ContactDetails, инфраструктуре Entity Framework Core сообщается о необходимости удаления временного объекта Supplier и снова вызывается метод SaveChanges(). Второе обновление приводит к удалению строки, созданной из объекта temp, и созданию отношения ContactDetails/Supplier, которое запросил пользователь.

Совет. Многоступенчатая операция подобного рода обычно выполняется внутри транзакции, так что изменения, примененные первым обновлением, могут быть автоматически отменены, если второе обновление в операции по какой-то причине потерпит неудачу. Работа транзакций и их поддержка инфраструктурой Entity Framework Core обсуждаются в главе 24.

Чтобы посмотреть, как все работает, перезапустите приложение, используя dotnet run, перейдите по ссылке http://localhost:5000/one2one и щелкните на одной из кнопок Edit. Выберите переключатель для объекта Supplier, который уже связан с объектом ContactDetails, и щелкните на кнопке Save. Код, добавленный к методу действия в листинге 16.13, обновит БД, поменяв отношения между объектами ContactDetails и Supplier (рис. 16.7).

Сначала выбирается объект ContactDetails для контакта Alice Jones, который связан с поставщиком Surf Dudes, и затем приложению сообщается о том, что взамен его нужно связать с поставщиком Acme Co. Поставщик Acme Co связан с объектом ContactDetails для контакта Bob Smith. Ниже перечислены действия, которые выполняются для обеспечения обновления, требующегося пользователю.

1. Изменить свойство SupplierId объекта Bob Smith на значение Id объекта Surf Dudes.
2. Создать временный объект Supplier и присвоить его свойству Supplier объекта Alice Jones.
3. Обновить БД, чтобы объект Acme Co не был связан отношением с объектом ContactDetails и стал "запасным".

4. Изменить свойство `SupplierId` объекта `Alice Jones` на значение `Id` объекта `Acme Co`.
5. Удалить временный объект `Supplier`.
6. Снова обновить БД, что сохранит отношение между объектами `Alice Jones` и `Acme Co` и удалит временный объект `Supplier`.

Несомненно, процесс выглядит неуклюжим (и вызывающим большую путаницу), но он гарантирует соблюдение ограничений в БД и отсутствие ошибок, генерируемых сервером баз данных.

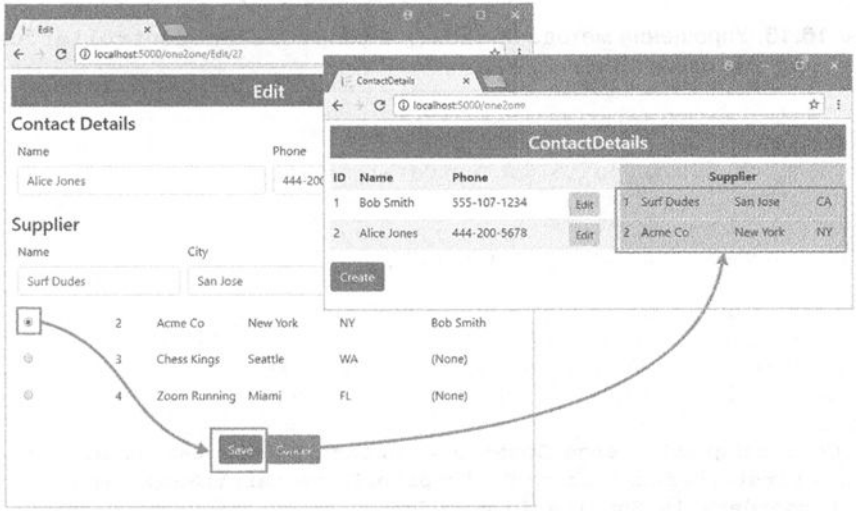

Рис. 16.7. Выбор поставщика, который уже связан с деталями о контакте

Изменение необязательного отношения "один к одному"

В случае необязательного отношения "один к одному" процесс проще, т.к. не нужно заботиться о том, чтобы каждый объект `ContactDetails` был связан с объектом `Supplier`, а только избегать дублирования значений внешнего ключа. Изменение типа свойства внешнего ключа на тип, допускающий значения `null`, изменит обязательное отношение на необязательное. Модифицируйте тип свойства внешнего ключа, определенного классом `ContactDetails`, как показано в листинге 16.14.

Листинг 16.14. Изменение типа свойства в файле `ContactDetails.cs` из папки `Models`

```
namespace DataApp.Models {
  public class ContactDetails {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Phone { get; set; }
    public ContactLocation Location { get; set; }
    public long? SupplierId { get; set; }
    public Supplier Supplier { get; set; }
  }
}
```

Выполните в папке проекта DataApp команды из листинга 16.15, чтобы обновить БД с целью отражения изменений в отношении. (Удалять и воссоздавать БД не требуется, потому что внесенное изменение ослабляет ограничения в ней.)

Листинг 16.15. Создание и применение миграции

```
dotnet ef migrations add OptionalOneToOne --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

После того как отношение стало необязательным, код в методе Update() контроллера можно упростить (листинг 16.16).

Листинг 16.16. Упрощение метода Update() в файле One2OneController.cs из папки Controllers

```
...
[HttpPost]
public IActionResult Update(ContactDetails details,
    long? targetSupplierId, long[] spares) {
  if (details.Id == 0) {
    context.Add<ContactDetails>(details);
  } else {
    context.Update<ContactDetails>(details);
    if (targetSupplierId.HasValue) {
      if (spares.Contains(targetSupplierId.Value)) {
        details.SupplierId = targetSupplierId.Value;
      } else {
        ContactDetails targetDetails = context.Set<ContactDetails>()
          .FirstOrDefault(cd => cd.SupplierId == targetSupplierId);
        targetDetails.SupplierId = null;
        details.SupplierId = targetSupplierId.Value;
        context.SaveChanges();
      }
    }
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Менять местами связанные объекты больше не требуется, поскольку теперь объект ContactDetails может храниться в БД без отношения с объектом Supplier. Когда пользователь выбирает объект Supplier, в БД отправляется запрос для нахождения объекта ContactDetails, с которым он связан, и его свойство SupplierId устанавливается в null. Шаг очень важен, т.к. значения SupplierId по-прежнему должны быть уникальными, а два объекта ContactDetails не могут быть связаны с одним и тем же объектом Supplier.

Чтобы увидеть эффект, перезапустите приложение, повторите процесс редактирования и выберите объект Supplier, который уже связан с каким-то объектом ContactDetails. Щелкнув на кнопке Save, вы увидите, что старое отношение разрывается, и объект ContactDetails становится несвязанным (рис. 16.8).

Рис. 16.8. Выбор связанного объекта `Supplier` в случае использования необязательного отношения

Определение отношений "многие ко многим"

Инфраструктуру Entity Framework Core можно применять для создания и управления отношениями "многие ко многим", где каждый объект одного типа может иметь неисключительные отношения с множеством объектов другого типа. Некоторые приложения не могут обойтись только отношениями "один к одному" и "один ко многим", поэтому в последующих разделах демонстрируется, как определять и использовать отношение "многие ко многим", несмотря на неуклюжую поддержку такой работы в Entity Framework Core.

В качестве подготовки к рассматриваемому далее примеру добавьте в папку `Models` файл класса по имени `Shipment.cs` с содержимым из листинга 16.17. Класс `Shipment` будет одним из участников отношения "многие ко многим".

Листинг 16.17. Содержимое файла `Shipment.cs` из папки `Models`

```
namespace DataApp.Models {

  public class Shipment {
    public long Id { get; set; }
    public string ShipperName {get; set;}
    public string StartCity { get; set; }
    public string EndCity { get; set; }
  }
}
```

Класс `Shipment` представляет поставку товаров. Для отслеживания того, какие товары были поставлены, будет создано отношение "многие ко многим" с классом `Product`. Оно означает, что объект `Product` может быть связан со многими объектами `Shipment` (указывая на доставку товаров для множества поставок), а объект `Shipment` — со многими объектами `Product` (указывая на возможность наличия в поставке множества товаров).

Создание соединяющего класса

Инфраструктура Entity Framework Core способна представлять отношение "многие ко многим" только за счет комбинирования двух отношений "один ко многим" и применения *соединяющего класса* для их связывания вместе. Не переживайте, если сказанное понятно не сразу, потому что оно станет яснее после того, как вы увидите, каким образом разные части сочетаются друг с другом.

Чтобы создать соединяющий класс, добавьте в папку `Models` файл класса по имени `ProductShipmentJunction.cs` с содержимым, показанным в листинге 16.18.

Листинг 16.18. Содержимое файла `ProductShipmentJunction.cs` из папки `Models`

```
namespace DataApp.Models {
  public class ProductShipmentJunction {
    public long Id { get; set; }
    public long ProductId { get; set; }
    public Product Product { get; set; }
    public long ShipmentId { get; set; }
    public Shipment Shipment { get; set; }
  }
}
```

Единственное предназначение соединяющего класса — служить контейнером для двух отношений "один ко многим". В классе `ProductShipmentJunction` определены два набора навигационных свойств и свойств внешнего ключа, которые создают отношения с классами `Product` и `Shipment` и зависимую сущность в обоих отношениях.

Укомплектование отношения "многие ко многим"

Укомплектование отношения означает добавление в классы `Product` и `Supplier` навигационных свойств, чтобы укомплектовать индивидуальные отношения "один ко многим" и сделать возможной навигацию с одного класса в другой через соединяющий класс. Добавьте навигационное свойство в класс `Product` (листинг 16.19).

Листинг 16.19. Добавление навигационного свойства в файле `Product.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {
  public enum Colors {
    Red, Green, Blue
  }
  public class Product {
    public long Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
    public Colors Color { get; set; }
    public bool InStock { get; set; }
```

```
    public long SupplierId { get; set; }
    public Supplier Supplier { get; set; }
    public IEnumerable<ProductShipmentJunction>
      ProductShipments { get; set; }
  }
}
```

Новое навигационное свойство укомплектовывает отношение с соединяющим классом, а не с классом `Shipment`. Добавьте в класс `Shipment` соответствующее свойство, которое укомплектовывает отношение с соединяющим классом (листинг 16.20).

Листинг 16.20. Добавление навигационного свойства в файле `Shipment.cs` из папки `Models`

```
using System.Collections.Generic;

namespace DataApp.Models {

  public class Shipment {
    public long Id { get; set; }
    public string ShipperName {get; set;}
    public string StartCity { get; set; }
    public string EndCity { get; set; }
    public IEnumerable<ProductShipmentJunction>
      ProductShipments { get; set; }
  }
}
```

В обоих сущностных классах определены навигационные свойства, которые возвращают объект реализации `IEnumerable<ProductShipmentJunction>` и не имеют прямого отношения друг с другом. Это усложняет навигационные операции и операции над данными, но основывается на знакомом фундаменте, предоставляя функциональную возможность "многие ко многим", как будет показано в последующих разделах.

Находясь в папке проекта `DataApp`, выполните команду из листинга 16.21 для создания миграции, которая обновит БД, чтобы она могла хранить объект `Shipment` и представлять отношение "многие ко многим".

Листинг 16.21. Создание миграции

```
dotnet ef migrations add ManyToMany --context EFDatabaseContext
```

Если вы просмотрите метод `Up()` в файле `<отметка времени>_ManyToMany.cs`, созданный в папке `Migrations`, то увидите, что были добавлены две новые таблицы. Инфраструктура Entity Framework Core обнаружила в классе `Product` новое навигационное свойство и определила, что оно необходимо для хранения объектов `ProductShipmentJunction` и `Shipment`. Класс `ProductShipmentJunction` является зависимой сущностью в обоих своих отношениях, и именно в эту таблицу Entity Framework Core добавляет столбцы внешнего ключа, которые будут использоваться для представления отношений с классами `Product` и `Supplier`. Результатом оказываются отношения "один ко многим" между классами `ProductShipmentJunction` и `Product`, а также между классами `ProductShipmentJunction` и `Shipment` (рис. 16.9).

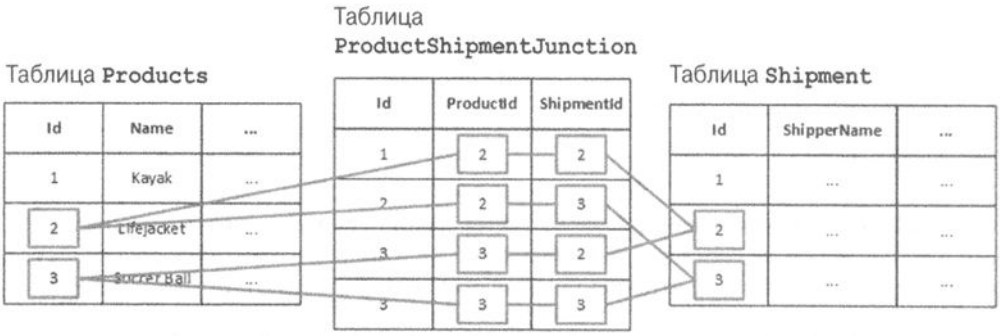

Рис. 16.9. Представление отношения "многие ко многим" в БД

Каждый объект `ProductShipmentJunction` действует как соединение между объектами `Product` и `Shipment`, а коллекция объектов `ProductShipmentJunction`, возвращаемая навигационными свойствами, предоставит доступ к полному набору связанных объектов, хотя и непрямо, как будет показано в последующих разделах.

Подготовка приложения

Чтобы получить в свое распоряжение данные для работы с ними, добавьте несколько объектов `Shipment` в класс заполнения БД начальными данными (листинг 16.22). Новые объекты не создают какие-либо отношения, и после сохранения в БД будут существовать в изоляции. Вскоре будет показано, как создавать и управлять отношениями.

Листинг 16.22. Добавление начальных данных в файле `SeedData.cs` из папки `Models`

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

namespace DataApp.Models {

  public static class SeedData {

    public static void Seed(DbContext context) {
      if (context.Database.GetPendingMigrations().Count() == 0) {
        if (context is EFDatabaseContext prodCtx
            && prodCtx.Products.Count() == 0) {
          prodCtx.Products.AddRange(Products);
          prodCtx.Set<Shipment>().AddRange(Shipments);
        } else if (context is EFCustomerContext custCtx
                && custCtx.Customers.Count() == 0) {
          custCtx.Customers.AddRange(Customers);
        }
        context.SaveChanges();
      }
    }

    public static void ClearData(DbContext context) {
      if (context is EFDatabaseContext prodCtx
          && prodCtx.Products.Count() > 0) {
        prodCtx.Products.RemoveRange(prodCtx.Products);
```

```
      prodCtx.Set<Shipment>()
        .RemoveRange(prodCtx.Set<Shipment>());
    } else if (context is EFCustomerContext custCtx
            && custCtx.Customers.Count() > 0) {
      custCtx.Customers.RemoveRange(custCtx.Customers);
    }
    context.SaveChanges();
  }
  public static Shipment[] Shipments {
    get {
      return new Shipment[] {
        new Shipment { ShipperName = "Express Co",
            StartCity = "New York", EndCity = "San Jose"},
        new Shipment { ShipperName = "Tortoise Shipping",
            StartCity = "Boston", EndCity = "Chicago"},
        new Shipment { ShipperName = "Air Express",
            StartCity = "Miami", EndCity = "Seattle"}
      };
    }
  }
  private static Product[] Products {
    get {
      // ...для краткости операторы не показаны...
    }
  }
  private static Customer[] Customers = {
    new Customer { Name = "Alice Smith",
        City = "New York", Country = "USA" },
    new Customer { Name = "Bob Jones",
        City = "Paris", Country = "France" },
    new Customer { Name = "Charlie Davies",
        City = "London", Country = "UK" }};
  }
}
```

Новое свойство `Shipments` возвращает массив объектов `Shipment`, которые добавляются в БД с применением метода `Set<T>().AddRange()` в методе `Seed()`. Метод `ClearData()` также был обновлен, чтобы удалять объекты `Shipment` из БД с использованием средства `Set<T>()`.

Начальные данные будут применяться только к пустой БД, поэтому выполните в папке проекта `DataApp` команды из листинга 16.23 для удаления и воссоздания БД со всеми миграциями, включая созданную в листинге 16.22, которая добавляет поддержку отношения "многие ко многим".

Листинг 16.23. Удаление и воссоздание БД

```
dotnet ef database drop --force --context EFDatabaseContext
dotnet ef database update --context EFDatabaseContext
```

Запрашивание связанных данных в отношении "многие ко многим"

Подход, требуемый для отношений "многие ко многим", влияет на способ выполнения запросов. При написании кода будьте внимательны, поскольку довольно легко сосредоточиться на классах, важных для приложения, и забыть о роли соединяющего класса.

Чтобы создать элементарный запрос, добавьте в папку `Controllers` файл класса по имени `Many2ManyController.cs` и определите в нем контроллер, как показано в листинге 16.24.

Листинг 16.24. Содержимое файла `Many2ManyController.cs` из папки `Controllers`

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace DataApp.Controllers {

  public class Many2ManyController : Controller {
    private EFDatabaseContext context;

    public Many2ManyController(EFDatabaseContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(new ProductShipmentViewModel {
        Products = context.Products.Include(p => p.ProductShipments)
          .ThenInclude(ps => ps.Shipment).ToArray(),
        Shipments = context.Set<Shipment>().Include(s => s.ProductShipments)
          .ThenInclude(ps => ps.Product).ToArray()
      });
    }
  }

  public class ProductShipmentViewModel {
    public IEnumerable<Product> Products { get; set; }
    public IEnumerable<Shipment> Shipments { get; set; }
  }
}
```

В контроллере `Many2Many` определен метод действия `Index()`, который передает объект `ProductShipmentViewModel` стандартному представлению. Класс `ProductShipmentViewModel` позволяет передавать представлению данные `Product` и `Shipment` согласованным образом.

Операторы запроса в действии `Index` заполняют данными свойства объектов модели представления, используя специальное свойство контекста для объектов `Product` и метод `Set<T>()` для запрашивания данных `Shipment`. В обоих случаях метод `Include()` применяется для следования по навигационному свойству в соединяющий класс, а метод `ThenInclude()` — для включения другого типа в отношение.

Соединяющий класс должен приниматься во внимание при отображении связанных данных в отношении такого вида. Чтобы снабдить представлением действие `Index`, определенное в листинге 16.24, создайте папку `Views/Many2Many` и добавьте в нее файл Razor по имени `Index.cshtml` с содержимым из листинга 16.25.

Листинг 16.25. Содержимое файла `Index.cshtml` из папки `Views/Many2Many`

```
@model DataApp.Controllers.ProductShipmentViewModel
@{
  ViewData["Title"] = "Many To Many";
  Layout = "_Layout";
}
<h4>Shipments</h4>
<table class="table table-sm table-striped">
  <tr><th>ID</th><th>Name</th><th>Product Names</th><th></th></tr>
  @if (Model.Shipments?.Count() > 0) {
    @foreach (Shipment s in Model.Shipments) {
      <tr>
        <td>@s.Id</td><td>@s.ShipperName</td>
        <td>
          @(string.Join(", ", s.ProductShipments
            .Select(ps => ps.Product.Name)))
        </td>
        <td>
          <a asp-action="EditShipment" asp-route-id="@s.Id"
             class="btn btn-sm btn-primary">Edit</a>
        </td>
      </tr>
    }
  } else {
    <tr><td colspan="3" class="text-center">No Data</td></tr>
  }
</table>
<h4>Products</h4>
<table class="table table-sm table-striped">
  <tr><th>ID</th><th>Name</th><th>Shipment Names</th></tr>
  @if (Model.Products?.Count() > 0) {
    @foreach (Product p in Model.Products) {
      <tr>
        <td>@p.Id</td><td>@p.Name</td>
        <td colspan="2">
          @(string.Join(", ", p.ProductShipments
            .Select(ps => ps.Shipment.ShipperName)))
        </td>
      </tr>
    }
  } else {
    <tr><td colspan="3" class="text-center">No Data</td></tr>
  }
</table>
```

Представление отображает таблицы, содержащие детали объектов `Product` и `Shipment`, а также ссылки, нацеленные на действия, которые будут добавлены позже в главе для демонстрации способа изменения отношений. Каждая строка таблицы использует навигационные свойства для достижения соединяющего класса, чтобы получить коллекцию связанных объектов, которые затем обрабатываются с применением LINQ и метода `string.Join()` с целью создания массива имен.

Совет. Представление оценивает последовательности объектов данных более одного раза, из-за чего для предотвращения отправки БД дублированных запросов в запросах в листинге 16.25 использовался метод `ToArray()`, как было описано в главе 15.

Запустите приложение с помощью `dotnet run` и перейдите по ссылке `http://localhost:5000/many2many`. Во время запуска БД заполнится начальными данными из листинга 16.25, но в текущий момент какие-либо отношения "многие ко многим" отсутствуют, что дает результат, показанный на рис. 16.10.

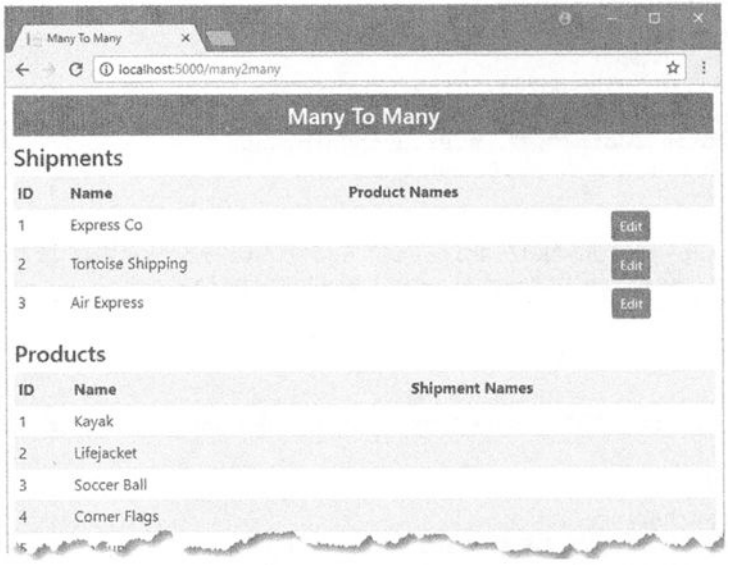

Рис. 16.10. Запрашивание данных в отношении "многие ко многим"

Управление отношениями "многие ко многим"

Чтобы приступить к реализации поддержки для управления отношениями, добавьте в контроллер `Many2Many` метод действия, приведенный в листинге 16.26. Этот метод действия запрашивает у БД объект, который пользователь выбрал, и передает его представлению по имени `ShipmentEditor`.

Листинг 16.26. Добавление действия в файле `Many2ManyController.cs` из папки `Controllers`

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;

namespace DataApp.Controllers {

  public class Many2ManyController : Controller {
    private EFDatabaseContext context;

    public Many2ManyController(EFDatabaseContext ctx) => context = ctx;
```

```
    public IActionResult Index() {
      return View(new ProductShipmentViewModel {
        Products = context.Products.Include(p => p.ProductShipments)
          .ThenInclude(ps => ps.Shipment).ToArray(),
        Shipments = context.Set<Shipment>().Include(s => s.ProductShipments)
          .ThenInclude(ps => ps.Product).ToArray()
      });
    }
    public IActionResult EditShipment (long id) {
      ViewBag.Products = context.Products.Include(p => p.ProductShipments);
      return View("ShipmentEditor", context.Set<Shipment>().Find(id));
    }
  }
  public class ProductShipmentViewModel {
    public IEnumerable<Product> Products { get; set; }
    public IEnumerable<Shipment> Shipments { get; set; }
  }
}
```

Для предоставления пользователю возможности редактировать отношения объекта `Shipment` нужен сам объект и полная коллекция объектов `Product`, чтобы можно было показать, какие из них уже связаны, а какие нет, что означает необходимость в получении также объектов `ProductShipmentJunction`. Объекты `Product` и `ProductShipmentJunction` получаются в единственном запросе, применяющем метод `Include()`, а результаты присваиваются свойству `ViewBag.Products`, которое обеспечивает доступ к объектам `Product` и средствам для выяснения, связаны ли они с объектом `Shipment`, выбранным пользователем. Для получения самого объекта `Shipment` используется метод `Set<T>()` объекта контекста БД, чтобы запросить БД с помощью метода `Find()`; объект `Shipment` передается в качестве модели методу `View()`.

Чтобы снабдить метод действия `EditShipment()` представлением, добавьте в папку `Views/Many2Many` файл представления по имени `ShipmentEditor.cshtml` с содержимым из листинга 16.27.

Листинг 16.27. Содержимое файла `ShipmentEditor.cshtml` из папки `Views/Many2Many`

```
@model DataApp.Models.Shipment
@{
  ViewData["Title"] = "Many To Many";
  Layout = "_Layout";
}
<div class="m-1 p-1">
  <div class="row">
    <div class="col"><strong>Name</strong></div>
    <div class="col"><strong>Start</strong></div>
    <div class="col"><strong>End</strong></div>
  </div>
  <div class="row">
    <div class="col">@Model.ShipperName</div>
    <div class="col">@Model.StartCity</div>
    <div class="col">@Model.EndCity</div>
  </div>
</div>
```

```
<form asp-action="UpdateShipment" method="post" class="p-2">
  <input type="hidden" name="id" value="@Model.Id" />
  <h4>Products</h4>
  @foreach (Product p in ViewBag.Products) {
    <div class="form-row">
      <div class="form-group col-1">
        @if (p.ProductShipments.Any(ps => ps.ShipmentId == Model.Id)) {
          <input type="checkbox" name="pids" value="@p.Id" checked />
        } else {
          <input type="checkbox" name="pids" value="@p.Id" />
        }
      </div>
      <div class="form-group col">
        <label class="form-control-label">@p.Name</label>
      </div>
      <div class="form-group col">
        <label class="form-control-label">@p.Category</label>
      </div>
      <div class="form-group col">
        <label class="form-control-label">@p.Price.ToString("C2")</label>
      </div>
    </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>
```

Объект `Shipment` применяется для отображения значений свойств пользователю и для предоставления значения скрытому элементу `input` в форме. Коллекция объектов `Product` используется для отображения сетки с детальной информацией, а объекты `ProductShipmentJunction` применяются для определения, связан ли конкретный объект `Product` с объектом `Shipment`.

Когда пользователь щелкает на кнопке Save, отображаемой представлением, HTML-форма отправляется действию по имени `UpdateShipment`. Добавьте в контроллер `Many2Many` метод действия `UpdateShipment()` с кодом, необходимым для обновления БД с целью отражения выбранных пользователем отношений (листинг 16.28).

На заметку! Код в листинге 16.28 обновляет только отношение "многие ко многим" и не затрагивает любые другие аспекты объекта `Shipment`. Демонстрация обновления объекта и его отношения "многие ко многим" в единственном методе действия приведена в главе 18.

Листинг 16.28. Обновление отношений в файле `Many2ManyController.cs` из папки `Controllers`

```
using DataApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
```

```
namespace DataApp.Controllers {
  public class Many2ManyController : Controller {
    private EFDatabaseContext context;

    public Many2ManyController(EFDatabaseContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(new ProductShipmentViewModel {
        Products = context.Products.Include(p => p.ProductShipments)
          .ThenInclude(ps => ps.Shipment).ToArray(),
        Shipments = context.Set<Shipment>().Include(s => s.ProductShipments)
          .ThenInclude(ps => ps.Product).ToArray()
      });
    }

    public IActionResult EditShipment (long id) {
      ViewBag.Products = context.Products.Include(p => p.ProductShipments);
      return View("ShipmentEditor", context.Set<Shipment>().Find(id));
    }

    public IActionResult UpdateShipment(long id, long[] pids) {
      Shipment shipment = context.Set<Shipment>()
        .Include(s => s.ProductShipments).First(s => s.Id == id);
      shipment.ProductShipments = pids.Select(pid
        => new ProductShipmentJunction {
            ShipmentId = id, ProductId = pid
          }).ToList();
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }

  public class ProductShipmentViewModel {
    public IEnumerable<Product> Products { get; set; }
    public IEnumerable<Shipment> Shipments { get; set; }
  }
}
```

Новый метод действия принимает значение свойства `Id` объекта `Shipment`, отредактированного пользователем, и массив значений `Id` для объектов `Product`, которым требуются отношения.

При обновлении отношений сначала у БД запрашивается объект `Shipment` вместе со связанными объектами `ProductShipmentJunction`. Это важный шаг, потому что если не извлечь объект `ProductShipmentJunction` из БД, то инфраструктура Entity Framework Core не удалит существующие отношения, которые больше не требуются.

Внимание! Если вы не запросите существующие отношения, то обнаружите, что новые отношения добавляются в БД, но отношения, выбор которых пользователь отменил, остаются неизмененными.

Следующий шаг — замена коллекции объектов `ProductShipmentJunction` коллекцией, содержащей только выбранные пользователем отношения:

```
...
shipment.ProductShipments = pids.Select(pid =>
  new ProductShipmentJunction { ShipmentId = id,
                                ProductId = pid }).ToList();
...
```

Метод `Select()` из LINQ используется для проецирования последовательности объектов `ProductShipmentJunction`, у которых свойства внешнего ключа установлены для представления одного из отношений, выбранных пользователем. (Метод `ToList()` позволяет создать коллекцию переменной длины, с которой рассчитывает работать инфраструктура Entity Framework Core.)

Для удаления существующих отношений, которые больше не нужны, никаких явных действий не требуется. Когда вызывается метод `SaveChanges()`, инфраструктура Entity Framework Core применит коллекцию объектов `ProductShipmentJunction`, созданную методом действия, для обновления БД, при необходимости создавая и удаляя данные.

Чтобы увидеть результат, перезапустите приложение, перейдите по ссылке `http://localhost:5000/many2many` и щелкните на кнопке Edit для одной из поставок. Отметьте объекты `Product`, для которых требуются отношения, и щелкните на кнопке Save. Отношения отразятся в разделах Shipments (Поставки) и Products (Товары) списка, причем каждый объект может быть связан с множеством других объектов (рис. 16.11).

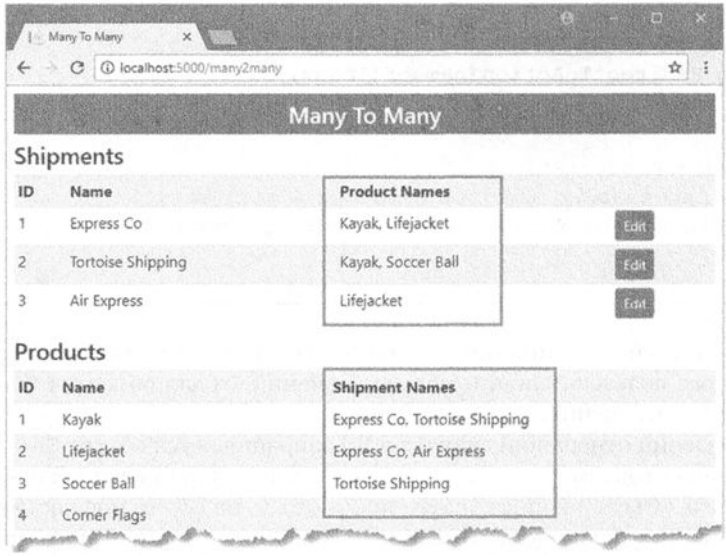

Рис. 16.11. Управление отношением "многие ко многим"

Резюме

В главе было показано, как инфраструктура Entity Framework Core поддерживает отношения "один к одному" и "многие ко многим". Вы узнали о том, как определять такие отношения, каким образом запрашивать связанные данные и как обновлять отношения по поручению пользователя. В следующей главе речь пойдет об использовании инфраструктуры Entity Framework Core с существующей БД.

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.

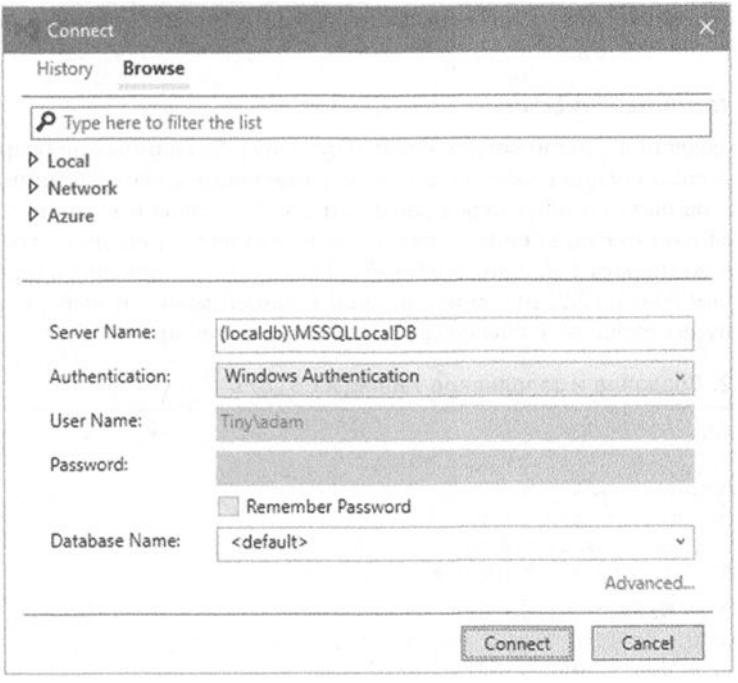

Рис. 17.1. Окно подключения к серверу баз данных

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.

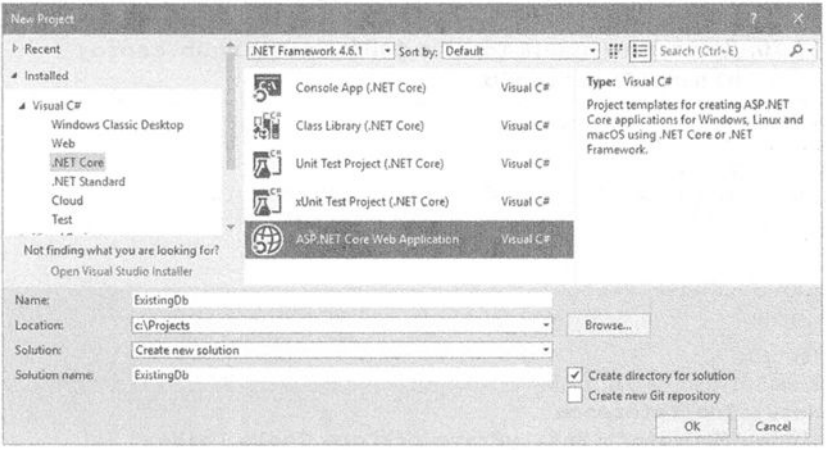

Рис. 17.2. Создание нового проекта ASP.NET Core MVC

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.

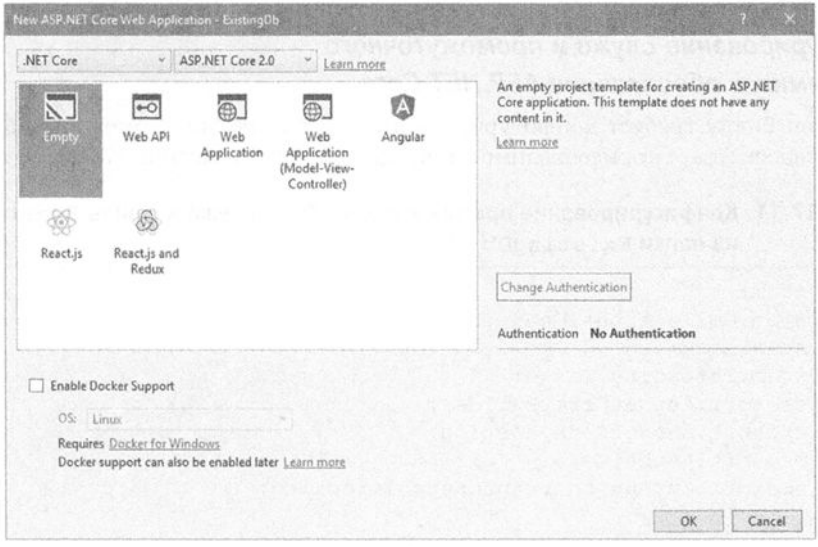

Рис. 17.3. Конфигурирование нового проекта ASP.NET Core MVC

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).

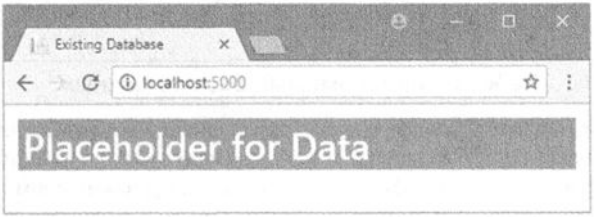

Рис. 17.4. Выполнение примера приложения

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();
    }
  }
}
```

Обновление контроллера и представления

Осталось обновить контроллер, чтобы он получал данные из БД и отображал их пользователю. Добавьте в контроллер Home операторы, которые запрашивают данные у БД (листинг 17.22). Планируется реализовать лишь несколько элементарных средств, поскольку в модель данных будут вноситься изменения, которые нежелательно распространять на множество методов действий и представлений.

Листинг 17.22. Запрашивание данных в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using ExistingDb.Models.Scaffold;
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Controllers {
  public class HomeController : Controller {
    private ScaffoldContext context;
    public HomeController(ScaffoldContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Shoes
        .Include(s => s.Color)
        .Include(s => s.SalesCampaigns)
        .Include(s => s.ShoeCategoryJunction)
          .ThenInclude(junct => junct.Category));
    }
  }
}
```

Работа с моделью данных, сгенерированной процессом формирования шаблонов, аналогична работе с моделью данных, которая создается с помощью миграции. В листинге 17.22 были выполнены запрос БД с применением свойства Shoes, определенного в классе контекста, и следование по навигационным свойствам к связанным данным посредством методов Include() и ThenInclude().

Чтобы включить вспомогательные функции дескрипторов ASP.NET Core MVC и предоставить легкий доступ к классам модели данных в представлениях, добавьте в папку Views файл по имени _ViewImports.cshtml с содержимым, показанным в листинге 17.23.

Листинг 17.23. Содержимое файла _ViewImports.cshtml из папки Views

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

Для отображения данных пользователю приведите содержимое файла Index.cshtml из папки Views/Home в соответствие с листингом 17.24.

Листинг 17.24. Отображение данных в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<Shoes>
@{
  ViewData["Title"] = "Existing Database";
  Layout = "_Layout";
}
```

```
<div class="container-fluid">
  <h4 class="bg-primary p-3 text-white">Zoom Shoes</h4>
  <table class="table table-striped">
    <thead>
      <tr>
        <th>Name</th><th>Price</th><th>Color</th>
        <th>Slogan</th><th>Categories</th>
      </tr>
    </thead>

    <tbody>
      @foreach (Shoes s in Model) {
        <tr>
          <td>@s.Name</td>
          <td>$@s.Price.ToString("F2")</td>
          <td>@s.Color.Name</td>
          <td>@s.SalesCampaigns?.Slogan</td>
          <td>
            @String.Join(", ", s.ShoeCategoryJunction
                  .Select(j => j.Category.Name))
          </td>
        </tr>
      }
    </tbody>

  </table>
</div>
```

Запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`; вы увидите таблицу, содержащую данные из БД, доступ к которым осуществлялся через модель данных, созданную с применением процесса формирования шаблонов (рис. 17.5).

Рис. 17.5. Использование модели данных, сгенерированной процессом формирования шаблонов

Реагирование на изменения в базе данных

Если вы пользуетесь существующей БД не единолично, тогда может возникнуть необходимость в реагировании на изменения, внесенные в интересах других приложений. В последующих разделах будет сымитировано изменение в БД и показано, как обновить модель данных, сгенерированную процессом формирования шаблонов, чтобы учесть изменение в приложении.

Модификация базы данных

Для эмуляции изменения в БД выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) и подключитесь к БД, как было описано в начале главы. Поместите SQL-код из листинга 17.25 в редактор запросов, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Execute (Выполнить).

Совет. При желании переустановить БД можете выполнить SQL-операторы из листингов 17.1–17.6. (Процесс будет проще, если вы загрузите проект для этой главы из хранилища GitHub, который содержит все необходимые SQL-файлы.)

Листинг 17.25. Изменение БД

```
USE ZoomShoesDb

CREATE TABLE Fittings (
    Id bigint IDENTITY(1,1) NOT NULL,
    Name nvarchar(max) NOT NULL,
  CONSTRAINT PK_Fittings PRIMARY KEY (Id));
GO

SET IDENTITY_INSERT Fittings ON
INSERT Fittings (Id, Name)
  VALUES (1, N'Narrow'),
         (2, N'Standard'),
         (3, N'Wide'),
         (4, N'Big Foot')
SET IDENTITY_INSERT Fittings OFF
GO

ALTER TABLE Shoes
  ADD FittingId bigint
ALTER TABLE Shoes
  ADD CONSTRAINT FK_Shoes_Fittings FOREIGN KEY(FittingId) REFERENCES
Fittings (Id)
GO

UPDATE Shoes SET FittingId = 2
GO

SELECT * from Shoes
```

Операторы из листинга 17.25 добавляют таблицу `Fittings` в БД и столбец внешнего ключа в таблицу `Shoes`, который ссылается на столбец первичного ключа в новой таблице. Последний оператор запрашивает таблицу `Shoes` и выдает данные, показанные в табл. 17.8.

Таблица 17.8. Эффект от добавления столбца в таблицу Shoes

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

Изменение такого типа вызывает проблемы в приложении ASP.NET Core MVC, потому что модель данных не имеет какого-либо представления столбца `OutOfStock`. Запросы к таблице `Shoes` не содержат все данные, а приложение не способно сохранять новые объекты в `Shoes` из-за отсутствия обязательного значения для столбца `OutOfStock`.

Обновление модели данных

Обновление модели данных с целью отражения изменений в БД означает повторение процесса формирования шаблонов. Выполните в папке проекта `ExistingDb` команду из листинга 17.26, удостоверившись в том, что в нее включены два дополнительных аргумента, выделенные полужирным.

Листинг 17.26. Обновление модели данных

```
dotnet ef dbcontext scaffold
"Server=(localdb)\MSSQLLocalDB;Database=ZoomShoesDb"
"Microsoft.EntityFrameworkCore.SqlServer"
--output-dir "Models/Scaffold"
--context ScaffoldContext --force --no-build
```

Аргумент `--force` сообщает инфраструктуре Entity Framework Core о необходимости замены существующих классов модели данных новыми классами. Аргумент `--no-build` предотвращает сборку проекта до выполнения процесса формирования шаблонов. Довольно легко попасть в ситуацию, когда процесс формирования шаблонов генерирует модель данных, которая не синхронизирована с остальными частями приложения, такими как контроллеры и представления. По умолчанию инфраструктура Entity Framework Core пытается собрать проект перед формированием шаблонов для БД, и неудавшаяся сборка — обычно из-за удаления свойства или класса модели данных — будет препятствовать выполнению процесса формирования шаблонов. Применение аргумента `--no-build` устраняет эту проблему и позволяет обновить остальные части приложения после того, как шаблоны для базы данных сформируются.

Обновление класса контекста

Процесс формирования шаблонов заменит класс контекста, перезаписав изменения, которые требуются для поддержки приложения ASP.NET Core MVC. После завершения процесса формирования шаблонов снова закомментируйте код метода `OnConfiguring()` и добавьте конструктор, получающий аргумент с параметрами конфигурации (листинг 17.27).

Листинг 17.27. 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<Fittings> Fittings { 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...
    }
  }
}
```

Обновление контроллеров и представлений

В зависимости от изменений, внесенных в БД, для обеспечения корректной работы приложения может понадобиться обновление остальных частей проекта. В рассматриваемом примере приложения задача сводится к обновлению контроллера Home для следования по новому навигационному свойству, добавленному в класс Shoes, с целью включения связанных данных в запросы, а также обновление представления для отображения этих данных. Изменение контроллера показано в листинге 17.28.

Листинг 17.28. Следование по новому навигационному свойству в файле HomeController.cs из папки Controllers

```
using Microsoft.AspNetCore.Mvc;
using ExistingDb.Models.Scaffold;
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Controllers {

  public class HomeController : Controller {
    private ScaffoldContext context;

    public HomeController(ScaffoldContext ctx) => context = ctx;
```

```
    public IActionResult Index() {
      return View(context.Shoes
        .Include(s => s.Color)
        .Include(s => s.SalesCampaigns)
        .Include(s => s.ShoeCategoryJunction)
          .ThenInclude(junct => junct.Category)
        .Include(s => s.Fitting));
    }
  }
}
```

Для отображения дополнительных данных пользователю к таблице в представлении `Index.cshtml` добавляется колонка (листинг 17.29).

Листинг 17.29. Отображение дополнительных данных в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Shoes>
@{
  ViewData["Title"] = "Existing Database";
  Layout = "_Layout";
}
<div class="container-fluid">
  <h4 class="bg-primary p-3 text-white">Zoom Shoes</h4>
  <table class="table table-striped">
    <thead>
      <tr>
        <th>Name</th>
        <th>Price</th>
        <th>Color</th>
        <th>Slogan</th>
        <th>Categories</th>
        <th>Fitting</th>
      </tr>
    </thead>
    <tbody>
      @foreach (Shoes s in Model) {
        <tr>
          <td>@s.Name</td>
          <td>$@s.Price.ToString("F2")</td>
          <td>@s.Color.Name</td>
          <td>@s.SalesCampaigns?.Slogan</td>
          <td>
            @String.Join(", ", s.ShoeCategoryJunction
                    .Select(j => j.Category.Name))
          </td>
          <td>@s.Fitting.Name</td>
        </tr>
      }
    </tbody>
  </table>
</div>
```

Чтобы увидеть эффект от изменения в БД, запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`; вы получите результат, показанный на рис. 17.6.

Рис. 17.6. Отображение данных из обновленной БД

Добавление возможностей постоянства модели данных

Многие модели данных являются просто классами, которые выступают в качестве коллекций данных и навигационных свойств для представления данных в БД. Но определенные приложения требуют наличия в классах модели данных дополнительного кода для реализации таких задач, как проверка достоверности или синтез данных, которые не доступны в БД. Повторное создание модели данных каждый раз, когда БД изменяется, может вызвать проблему, поскольку классы модели перезаписываются, из-за чего дополнительная логика утрачивается.

Чтобы решить проблему, при построении модели данных во время процесса формирования шаблонов инфраструктура Entity Framework Core создает частичные классы. Например, вот определение класса `Shoes`, созданного инфраструктурой Entity Framework Core:

```
...
public partial class Shoes {
...
```

Частичные классы можно определять в нескольких файлах, что открывает возможность определения любой дополнительной логики, требующейся приложению, отдельно от файла, который создается процессом формирования шаблонов, и гарантирует ее существование после внесения изменений в БД. В качестве демонстрации создайте папку `Models/Logic` и поместите в нее файл класса по имени `Shoes.cs` с кодом из листинга 17.30.

Листинг 17.30. Содержимое файла `Shoes.cs` из папки `Models/Logic`

```
namespace ExistingDb.Models.Scaffold {
  public partial class Shoes {
    public decimal PriceIncTax => this.Price * 1.2m;
  }
}
```

Разные части частичного класса должны быть определены в том же самом пространстве имен, и это причина нахождения этого класса в пространстве имен `ExistingDb.Models.Scaffold`, хотя файл класса помещен в папку `Logic`. Одна часть частичного класса может обращаться к членам из остальных частей, т.е. свойство `PriceIncTax` способно читать значение свойства `Price`.

Внимание! Определение дополнительной логики для классов модели в частичном классе означает, что при повторном создании модели данных код не затрагивается, но вы по-прежнему должны удостоверяться в пригодности и правильности кода после изменения в БД.

В листинге 17.31 обновляется представление, которое отображает данные, с целью применения свойства, определенного в листинге 17.30.

Листинг 17.31. Использование свойства, определенного внутри частичного класса, в файле `Index.cshtml` из папки `Views/Home`

```
...
<tbody>
  @foreach (Shoes s in Model) {
    <tr>
      <td>@s.Name</td>
      <td>$@s.PriceIncTax.ToString("F2")</td>
      <td>@s.Color.Name</td>
      <td>@s.SalesCampaigns?.Slogan</td>
      <td>
        @String.Join(", ", s.ShoeCategoryJunction
              .Select(j => j.Category.Name))
      </td>
      <td>@s.Fitting.Name</td>
    </tr>
  }
</tbody>
...
```

Для просмотра эффекта от дополнительной логики запустите приложение с помощью команды `dotnet run` и перейдите по ссылке `http://localhost:5000`, что отобразит цены с 20%-ным увеличением, учитывающим налог (рис. 17.7).

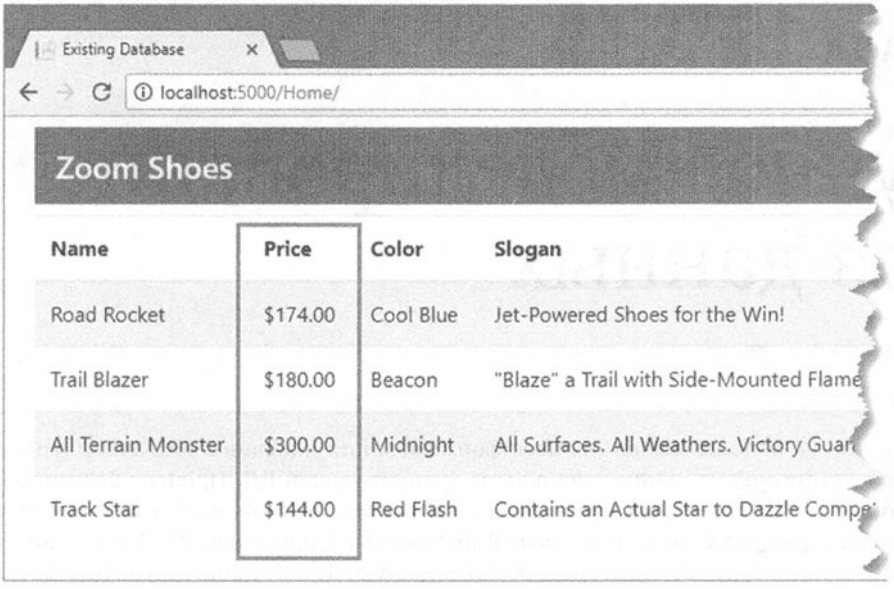

Рис. 17.7. Применение частичного класса для реализации дополнительной логики модели

Резюме

В главе было показано, как использовать существующую БД с применением средства формирования шаблонов Entity Framework Core, которое инспектирует БД и создает классы контекста и сущностей, требующиеся для работы с ней. Средство формирования шаблонов удобно для простых БД, но в более сложных проектах использовать его труднее. В следующей главе объясняется, каким образом создавать модель данных вручную, не полагаясь на процесс формирования шаблонов.

ГЛАВА 18

Ручное моделирование баз данных

В главе 17 использовался процесс формирования шаблонов для автоматического создания модели данных на основе существующей БД. Процесс формирования шаблонов удобен, но он не предлагает особо много возможностей в плане точного контроля, а результат может оказаться неудобным в применении. В настоящей главе будет показано, как моделировать БД вручную. Такой процесс требует большего объема усилий, но производит модель данных, с которой более естественно работать и которой легче управлять, когда в БД вносятся изменения. В табл. 18.1 приведены сведения, позволяющие поместить ручное моделирование баз данных в контекст.

Таблица 18.1. Помещение ручного моделирования в контекст

Вопрос	Ответ
Что это такое?	Ручное моделирование создает модель данных для БД, не используя формирование шаблонов
Чем оно полезно?	Средство формирования шаблонов не всегда справляется с крупными и сложными БД
Как оно используется?	Для конфигурирования модели данных в классе контекста применяется интерфейс Fluent API
Существуют ли какие-то скрытые ловушки или ограничения?	Если вы моделируете только часть БД, то должны удостовериться, что она не имеет зависимостей от частей, которые были исключены
Существуют ли альтернативы?	Единственная альтернатива — использование средства формирования шаблонов

В табл. 18.2 приведена сводка по главе.

Таблица 18.2. Сводка по главе

Задача	Решение	Листинг
Моделирование БД вручную	Создайте класс контекста и примените интерфейс Fluent API для конфигурирования модели данных	18.1–18.11, 18.19–18.32
Моделирование отношения вручную	Определите навигационные свойства и сконфигурируйте отношения с использованием атрибутов или операторов Fluent API	18.12–18.18

Подготовительные шаги

В главе продолжается работа с проектом `ExistingDb` и БД, созданными в главе 17. Если вы пропустили главу 17, тогда необходимо выполнить шаги, описанные в ее начале, чтобы создать БД, на которую полагаются примеры в текущей главе.

На заметку! Вы можете загрузить проект, который содержит файлы с SQL-операторами, необходимыми для создания БД, из хранилища исходного кода для книги по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Чтобы обеспечить создание БД и наличие в ней ожидаемых данных, запустите приложение с помощью команды `dotnet run` и перейдите по ссылке `http://localhost:5000`; результат приведен на рис. 18.1.

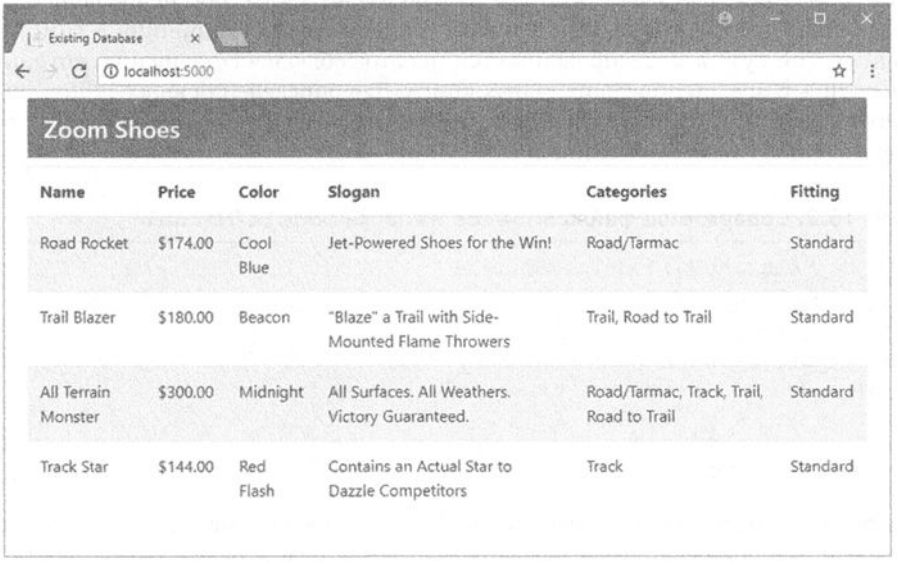

Рис. 18.1. Выполнение примера приложения

Создание ручной модели данных

Прежде чем приступать к моделированию существующей БД, вы должны понять ее схему и выяснить, какие части БД нужны для приложения ASP.NET Core MVC. В отсутствие подробного описания БД и ее структуры хорошей отправной точкой может послужить выполнение процесса формирования шаблонов, рассмотренного в главе 17, даже если созданная в результате него модель данных будет применяться лишь в справочных целях при построении ручной модели.

Создание класса контекста и сущностных классов

Первым шагом в создании модели данных вручную будет определение класса контекста. Создайте папку `Models/Manual` и добавьте в нее файл класса по имени `ManualContext.cs` с содержимым из листинга 18.1.

Листинг 18.1. Содержимое файла `ManualContext.cs` из папки `Models/Manual`

```
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Models.Manual {

  public class ManualContext : DbContext {

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

    public DbSet<Shoe> Shoes { get; set; }
  }
}
```

Класс `ManualContext` является производным от класса `DbContext`, имеет конструктор, который принимает объект конфигурации и передает его конструктору суперкласса, и определяет свойство `DbSet<T>` по имени `Shoes`, обеспечивающее доступ к коллекции объектов `Shoe`. Если это кажется знакомым, то причина в том, что вы прошли долгий путь к моделированию БД, просто соблюдая соглашения, которые использовались в предшествующих главах книги. Для определения класса `Shoe`, применяемого контекстом, добавьте в папку `Models/Manual` файл класса по имени `Shoe.cs` и поместите в него код, показанный в листинге 18.2.

Листинг 18.2. Содержимое файла `Shoe.cs` из папки `Models/Manual`

```
namespace ExistingDb.Models.Manual {

  public class Shoe {
    public long Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
  }
}
```

Класс `Shoe` содержит свойства для ряда столбцов в таблице `Shoes`, но навигационные свойства пока еще не определены. Чтобы сделать новый контекст доступным для использования в остальных частях приложения, создайте службу, как демонстрируется в листинге 18.3.

Листинг 18.3. Создание службы 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;
using ExistingDb.Models.Manual;
```

```
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));
      services.AddDbContext<ManualContext>(options =>
        options.UseSqlServer(conString));
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Новый контекст применяет ту же самую строку подключения, что и контекст, созданный процессом формирования шаблонов, поскольку они оба подключаются в той же БД.

Создание контроллера и представления

Чтобы протестировать начальную модель данных, создайте в папке `Controllers` файл класса по имени `ManualController.cs` с кодом из листинга 18.4.

Листинг 18.4. Содержимое файла `ManualController.cs` из папки `Controllers`

```
using ExistingDb.Models.Manual;
using Microsoft.AspNetCore.Mvc;

namespace ExistingDb.Controllers {
  public class ManualController : Controller {
    private ManualContext context;
    public ManualController(ManualContext ctx) => context = ctx;
    public IActionResult Index() => View(context.Shoes);
  }
}
```

Метод действия `Index()` передает объект, возвращаемый свойством `Shoes` контекста, стандартному представлению. Чтобы снабдить действие его представлением, создайте папку `Views/Manual` и поместите в нее файл представления, содержимое которого приведено в листинге 18.5.

Листинг 18.5. Содержимое файла `Index.cshtml` из папки `Views/Manual`

```
@using ExistingDb.Models.Manual
@model IEnumerable<Shoe>
@{
  ViewData["Title"] = "Manual Data Model";
  Layout = "_Layout";
}
<div class="container-fluid">
  <h4 class="bg-primary p-3 text-white">Zoom Shoes</h4>
  <table class="table table-striped table-sm">
    <tr><th>Id</th><th>Name</th><th>Price</th></tr>
    @foreach (Shoe s in Model) {
      <tr>
        <td>@s.Id</td>
        <td>@s.Name</td>
        <td>$@s.Price.ToString("F2")</td>
      </tr>
    }
  </table>
</div>
```

Представление отображает таблицу, которая содержит значения `Id`, `Name` и `Price` для каждого объекта `Shoe` в последовательности, полученной от метода действия. Чтобы протестировать ручную модель данных, запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000/manual`, что приведет к получению результата, показанного на рис. 18.2.

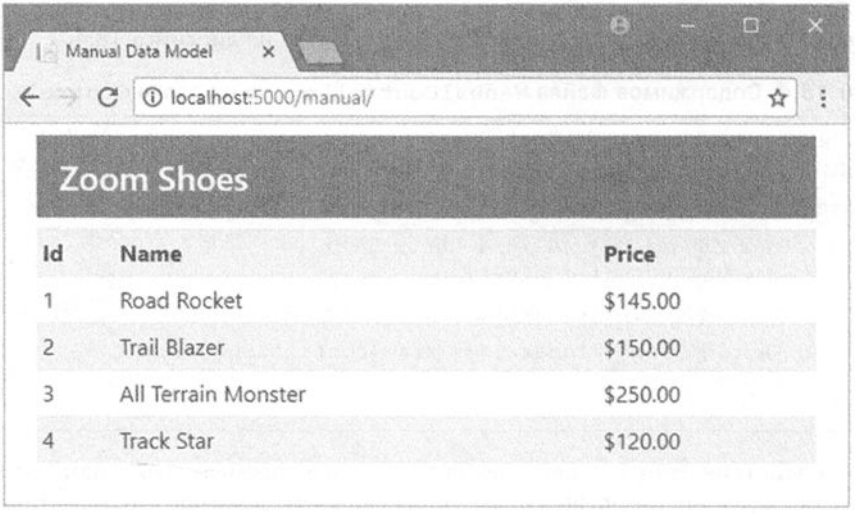

Рис. 18.2. Тестирование ручной модели данных

Основные соглашения для модели данных

Как было продемонстрировано в предыдущем разделе, создавать модель данных вручную легко, если вы способны следовать соглашениям, которые ожидает инфраструктура Entity Framework Core. Даже в простой модели данных вроде созданной в предыдущем разделе соблюдалось несколько соглашений для получения данных из БД. Каждое соглашение будет объясняться в свое время, но главное внимание сосредоточено на следующих наиболее основных соглашениях.

- Имя свойства в классе контекста соответствует имени таблицы в БД, так что свойство `Shoes` в классе контекста соответствует таблице `Shoes` в БД. (Если свойство в классе контекста не предусмотрено и доступ к данным осуществляется через метод `Set<T>()`, тогда в качестве имени таблицы применяется имя сущностного класса.)
- Имена свойств в сущностном классе соответствуют именам столбцов в таблице БД, так что свойства `Id`, `Name` и `Price` используются для представления значений столбцов `Id`, `Name` и `Price` в таблице `Shoes`.
- Первичный ключ представляется свойством по имени `Id` или `<Тип>Id`, так что первичный ключ для класса `Shoe` определяется как свойство `Id` или `ShoeId`.

К описанным соглашениям вы уже привыкли в предшествующих главах, когда БД создавалась из класса контекста и сущностных классов с применением миграций. При работе с существующей БД инфраструктура Entity Framework Core использует такие же соглашения, а это значит, что отображение между объектами и БД может быть настроено без необходимости в явной конфигурации.

Переопределение соглашений для модели данных

Соглашения Entity Framework Core удобны, когда структура БД совпадает с тем, что нужно в приложении. Но так случается редко, особенно если предпринимается попытка интеграции в проект существующей БД. Инфраструктура Entity Framework Core предлагает два способа, которыми можно переопределить соглашения, что позволяет создавать модель данных, подходящую для части ASP.NET Core MVC проекта, и одновременно предоставлять доступ к данным в БД: атрибуты и Fluent API. Атрибуты применять легко, но они не открывают доступ к полному спектру возможностей Entity Framework Core. Интерфейс Fluent API сложнее, но он обеспечивает больший контроль над тем, как модель данных отображается на БД. В последующих разделах будут описаны оба подхода вместе с демонстрацией их использования для переопределения упомянутых ранее трех основных соглашений.

Использование атрибутов для переопределения соглашений, касающихся модели данных

Основные соглашения для модели данных могут быть переопределены с помощью атрибутов, перечисленных в табл. 18.3, которые применяются к сущностным классам модели.

Добавьте в папку `Models/Manual` файл по имени `Style.cs` и определите в нем класс, как показано в листинге 18.6, для демонстрации работы описанных атрибутов.

Таблица 18.3. Атрибуты для переопределения соглашений, связанных с моделью данных

Имя	Описание
`Table`	Указывает таблицу БД и переопределяет имя свойства в классе контекста
`Column`	Указывает столбец, предоставляющий значения для свойства, к которому атрибут применен
`Key`	Используется для идентификации свойства, которому будет присвоено значение первичного ключа

Листинг 18.6. Содержимое файла `Style.cs` из папки `Models/Manual`

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace ExistingDb.Models.Manual {
  [Table("Colors")]
  public class Style {
    [Key]
    [Column("Id")]
    public long UniqueIdent { get; set; }
    [Column("Name")]
    public string StyleName { get; set; }
    public string MainColor { get; set; }
    public string HighlightColor { get; set; }
  }
}
```

Атрибут `Table` сообщает инфраструктуре Entity Framework Core о том, что источником данных для класса `Style` является таблица `Colors`. Атрибут `Key` указывает, что свойство `UniqueIdent` должно применяться для значений первичного ключа, и заданный вместе с ним атрибут `Column` гарантирует, что в качестве источника таких значений будет использоваться столбец `Id`. Атрибут `Column` сообщает инфраструктуре Entity Framework Core о том, что свойству `StyleName` должны присваиваться значения из столбца `Name`.

Атрибуты должны применяться только для требующихся изменений, что позволяет полагаться на соглашения в случае свойств `MainColor` и `HighlightColor`, которым будут присваиваться значения из столбцов с такими же именами. Чтобы завершить пример, добавьте в класс контекста свойство, сделав доступ к данным более удобным (листинг 18.7).

Листинг 18.7. Добавление свойства в файле `ManualContext.cs` из папки `Models/Manual`

```
using Microsoft.EntityFrameworkCore;
namespace ExistingDb.Models.Manual {
  public class ManualContext : DbContext {
    public ManualContext(DbContextOptions<ManualContext> options)
      : base(options) { }
    public DbSet<Shoe> Shoes { get; set; }
    public DbSet<Style> ShoeStyles { get; set; }
  }
}
```

Свойство `ShoeStyles` будет использоваться для доступа к данным в следующем разделе после представления интерфейса Fluent API.

Выбор между атрибутами и Fluent API

Основные соглашения для модели данных могут быть переопределены с применением либо атрибутов, либо Fluent API, поэтому вы вправе выбрать тот подход, который воспринимаете как наиболее естественный. Одни разработчики предпочитают аннотировать классы с помощью атрибутов, потому что это согласуется с тем, как работают средства ASP.NET Core MVC, подобные проверке достоверности и авторизации. Другие разработчики предпочитают описывать модель данных в классе контекста, чтобы все изменения обычных соглашений были собраны в одном месте.

Некоторыми расширенными средствами можно пользоваться только через интерфейс Fluent API. Когда такие средства нужны для моделирования БД, то нет иного выбора, кроме как применять Fluent API, даже если вы предпочитаете работать с атрибутами. Тем не менее, как будет показано в главе, при создании модели данных атрибуты и Fluent API можно смешивать, используя атрибуты для тех средств, которые их поддерживают, если вы отдаете им предпочтение. В таком случае не забывайте, что операторы Fluent API имеют более высокий приоритет, чем атрибуты, которые будут молча игнорироваться, если вы переопределите то же самое соглашение с применением Fluent API.

Использование Fluent API для переопределения соглашений, касающихся модели данных

Интерфейс Fluent API применяется для переопределения соглашений, связанных с моделью данных, за счет описания частей модели данных программным путем. Атрибуты пригодны при внесении простых изменений, но со временем возникнет ситуация, когда подходящего атрибута нет, и требуется расширенное средство, которое поддерживается только в Fluent API.

На заметку! В этой главе интерфейс Fluent API используется для моделирования существующей БД, но его также можно применять для точной настройки миграций в приложении, разработанном в стиле "сначала код", как демонстрируется в третьей части книги.

Для создания примера, эквивалентного примеру использования атрибутов в предыдущем разделе, добавьте в папку `Models/Manual` файл по имени `ShoeWidth.cs`, содержимое которого приведено в листинге 18.8.

Листинг 18.8. Содержимое файла `ShoeWidth.cs` из папки `Models/Manual`

```
namespace ExistingDb.Models.Manual {
  public class ShoeWidth {
    public long UniqueIdent { get; set; }
    public string WidthName { get; set; }
  }
}
```

Класс `ShoeWidth` будет применяться для представления данных в таблице `Fittings`. В нем не соблюдаются соглашения, принятые в Entity Framework Core: имя класса не совпадает с именем таблицы БД, а для столбцов `Id` и `Name` определены свойства `UniqueIdent` и `WidthName`.

Вместо того чтобы модифицировать сущностный класс, в классе контекста переопределяется метод `OnModelCreating()` с использованием Fluent API (листинг 18.9).

Листинг 18.9. Применение Fluent API в файле `ManualContext.cs` из папки `Models/Manual`

```
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Models.Manual {

  public class ManualContext : DbContext {

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

    public DbSet<Shoe> Shoes { get; set; }

    public DbSet<Style> ShoeStyles { get; set; }

    public DbSet<ShoeWidth> ShoeWidths { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<ShoeWidth>().ToTable("Fittings");
      modelBuilder.Entity<ShoeWidth>().HasKey( t => t.UniqueIdent);
      modelBuilder.Entity<ShoeWidth>()
        .Property(t => t.UniqueIdent)
        .HasColumnName("Id");

      modelBuilder.Entity<ShoeWidth>()
        .Property(t => t.WidthName)
        .HasColumnName("Name");
    }
  }
}
```

Метод `OnModelCreating()` принимает объект `ModelBuilder`, на котором используется Fluent API. Самым важным методом, определенным в классе `ModelBuilder`, является `Entity<T>()`, который позволяет описывать сущностный класс для инфраструктуры Entity Framework Core и переопределять соглашения, применяемые в противном случае.

Метод `Entity<T>()` возвращает объект `EntityTypeBuilder<T>`, определяющий набор методов, которые используются при описании модели данных для Entity Framework Core. В табл. 18.4 приведены методы `EntityTypeBuilder<T>`, задействованные в листинге 18.9.

Совет. Методы Fluent API, выбирающие свойство сущностного класса, такие как `HasKey()` и `Property()`, перегружены для того, чтобы свойства можно было указывать с помощью строк или лямбда-выражения. Лямбда-выражения позволяют избегать опечаток, которые приводят к ошибкам, обнаруживаемым лишь во время выполнения приложения, поэтому именно такие методы применяются в книге.

Таблица 18.4. Методы `EntityTypeBuilder<T>`, использованные в листинге 18.9

Имя	Описание
`ToTable(table)`	Позволяет указать таблицу для сущностного класса; эквивалент атрибута `Table`
`HasKey(selector)`	Позволяет указать свойство ключа для сущностного класса; эквивалент атрибута `Key`. Аргументом является лямбда-выражение, которое выбирает свойство ключа
`Property(selector)`	Позволяет выбрать свойство, как будет описано ниже

Методы `ToTable()` и `HasKey()` применялись для указания таблицы БД и свойства первичного ключа для класса `ShoeWidth`. Метод `Property()` используется для выбора свойства с целью дальнейшего конфигурирования и возвращает объект `PropertyBuilder<T>`, где `T` - тип, возвращаемый выбранным свойством. В классе `PropertyBuilder<T>` определено несколько методов, которые предоставляют точный контроль над свойством и будут описаны далее в этой и последующих главах. В листинге 18.9 применялся метод `PropertyBuilder<T>` по имени `HasColumnName()`, описанный в табл. 18.5 и предназначенный для указания столбца таблицы БД, которая будет снабжать значениями выбранное свойство.

Таблица 18.5. Метод `PropertyBuilder<T>`, использованный в листинге 18.9

Имя	Описание
`HasColumnName(name)`	Применяется для выбора столбца, который будет снабжать значениями выбранное свойство; эквивалент атрибута `Column`

Использование настроенной модели данных

Независимо от того, применяете вы атрибуты или Fluent API, после переопределения соглашений для создания модели данных, требующейся приложению, вы можете использовать класс контекста и сущностные классы обычным образом. В завершение этого раздела добавьте в метод действия `Index()` контроллера `Manual` операторы, которые передают объекты `ShoeStyle` и `ShoeWidth` стандартному представлению посредством `ViewBag` (листинг 18.10).

Листинг 18.10. Работа с настроенной моделью данных в файле `ManualController.cs` из папки `Controllers`

```
using ExistingDb.Models.Manual;
using Microsoft.AspNetCore.Mvc;

namespace ExistingDb.Controllers {
  public class ManualController : Controller {
    private ManualContext context;
    public ManualController(ManualContext ctx) => context = ctx;
    public IActionResult Index() {
      ViewBag.Styles = context.ShoeStyles;
      ViewBag.Widths = context.ShoeWidths;
      return View(context.Shoes);
    }
  }
}
```

Для отображения данных пользователю добавьте в представление `Index.cshtml` из папки `Views/Manual` разметку, выделенную полужирным в листинге 18.11.

Листинг 18.11. Отображение дополнительных данных в файле `Index.cshtml` из папки `Views/Manual`

```
@using ExistingDb.Models.Manual
@model IEnumerable<Shoe>
@{
  ViewData["Title"] = "Manual Data Model";
  Layout = "_Layout";
}
<div class="container-fluid">
  <h4 class="bg-primary p-3 text-white">Zoom Shoes</h4>
  <table class="table table-striped table-sm">
    <tr><th>Id</th><th>Name</th><th>Price</th></tr>
    @foreach (Shoe s in Model) {
      <tr>
        <td>@s.Id</td>
        <td>@s.Name</td>
        <td>$@s.Price.ToString("F2")</td>
      </tr>
    }
  </table>
  <div class="row">
    <div class="col">
      <h5 class="bg-primary p-2 text-white">Styles</h5>
      <table class="table table-striped table-sm">
        <tr>
          <th>UniqueIdent</th><th>Style Name</th>
          <th>Main Color</th><th>Highlight Color</th>
        </tr>
        @foreach (Style s in ViewBag.Styles) {
          <tr>
            <td>@s.UniqueIdent</td>
            <td>@s.StyleName</td>
            <td>@s.MainColor</td>
            <td>@s.HighlightColor</td>
          </tr>
        }
      </table>
    </div>
    <div class="col">
      <h5 class="bg-primary p-2 text-white">Widths</h5>
      <table class="table table-striped table-sm">
        <tr><th>UniqueIdent</th><th>Name</th></tr>
        @foreach (ShoeWidth s in ViewBag.Widths) {
          <tr><td>@s.UniqueIdent</td><td>@s.WidthName</td></tr>
        }
      </table>
    </div>
  </div>
</div>
```

Чтобы протестировать оба приема переопределения соглашений, касающихся модели данных, запустите приложение с применением команды `dotnet run` и перейдите по ссылке `http://localhost:5000/manual`. Во время запуска при построении модели данных инфраструктура Entity Framework Core будет использовать атрибуты и операторы Fluent API, так что запросы, выполненные методом действия, гладко отображаются на БД, давая показанный на рис. 18.3 результат.

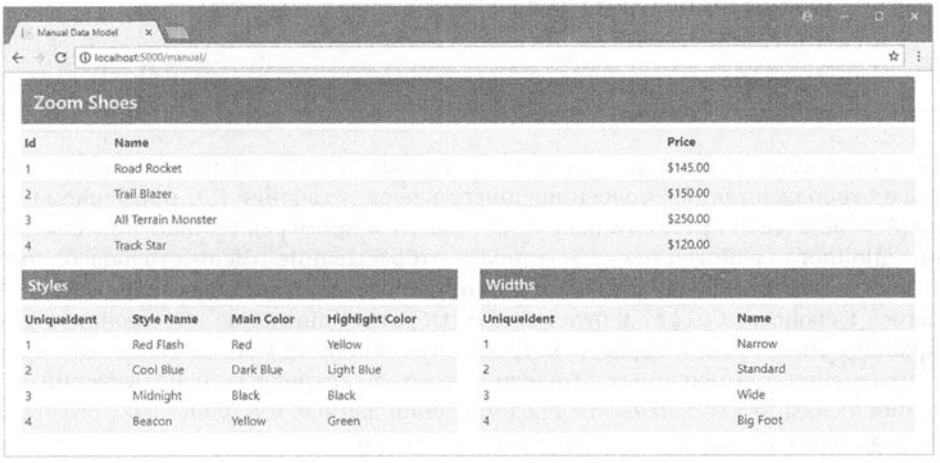

Рис. 18.3. Переопределение модели данных

Реагирование на изменения в базе данных

В случае внесения изменений в БД вы должны обеспечить обновление модели данных. Какого-либо автоматического процесса обновления вроде того, что применялся со средством формирования шаблонов, описанным в главе 17, не предусмотрено.

Если вы не обновите модель данных надлежащим образом, то создадите несоответствие между Entity Framework Core и БД. Проблема подобного рода не станет явной вплоть до стадии выполнения и может вызвать тонкие затруднения, которые проявятся только для специфических операций или определенных значений данных.

Удостоверьтесь в том, что понимаете влияние изменений в БД на приложение и придерживайтесь режима тестирования, который сводит к минимуму вероятность возникновения ошибок в производственных системах.

Моделирование отношений

Несмотря на создание трех классов для представления данных в БД, каждый из них существует обособленно и должен запрашиваться индивидуально. Определение отношений между классами означает добавление навигационных свойств и свойств внешнего ключа, следуя тем же соглашениям, которые использовались при разработке в стиле "сначала код". Добавьте в класс `Shoes` навигационное свойство и свойство внешнего ключа, которые моделируют сторону "один" отношения "один ко многим" между таблицами `Shoes` и `Colors` в БД (листинг 18.12).

Листинг 18.12. Добавление свойств в файле `Shoes.cs` из папки `Models/Manual`

```
namespace ExistingDb.Models.Manual {
  public class Shoe {
    public long Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public long ColorId { get; set; }
    public Style Color { get; set; }
  }
}
```

При моделировании существующей БД самым важным свойством является то, которое отображается на столбец внешнего ключа в таблице БД, применяемой для хранения зависимой сущности (класса `Shoe` в этом примере). Соглашение для имени навигационного свойства предусматривает отбрасывание имени столбца из имени свойства внешнего ключа, так что навигационным свойством для отношения, хранящегося в свойстве `ColorId`, будет `Color`. Обратите внимание, что свойство `Color` возвращает тип `Style`; инфраструктура Entity Framework Core согласованно применяет изменения, примененные к модели данных, т.е. атрибуты, использованные для указания класса `Style` в качестве представления данных в таблице `Colors`, продолжают действовать даже при определении отношений.

Чтобы укомплектовать отношение, добавьте в класс `Style` обратное навигационное свойство (листинг 18.13).

Листинг 18.13. Укомплектование отношения в файле `Style.cs` из папки `Models/Manual`

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Collections.Generic;

namespace ExistingDb.Models.Manual {
  [Table("Colors")]
  public class Style {
    [Key]
    [Column("Id")]
    public long UniqueIdent { get; set; }
    [Column("Name")]
    public string StyleName { get; set; }
    public string MainColor { get; set; }
    public string HighlightColor { get; set; }
    public IEnumerable<Shoe> Shoes { get; set; }
  }
}
```

Переопределение соглашений, касающихся отношений, с использованием атрибутов

Отношение, определенное в листинге 18.12, раскрывает внутреннюю структуру БД через изменения, которые были применены к модели, с тем результатом, что для получения объекта `Style` нужно проследовать по свойству с именем `Color`.

Недостаток переопределения соглашений, касающихся модели данных, связан с тем, что однажды начав, вам придется продолжать работу с изменениями на всем пути, в том числе обеспечивать согласованность имен навигационных свойств и свойств внешнего ключа с другими изменениями, которые вы внесли. Переопределите стандартные соглашения, касающиеся отношений, для свойства внешнего ключа и навигационного свойства в классе `Shoe` с использованием атрибутов (листинг 18.14).

Листинг 18.14. Переопределение соглашений в файле `Shoe.cs` из папки `Models/Manual`

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ExistingDb.Models.Manual {
  public class Shoe {
    public long Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    [Column("ColorId")]
    public long StyleId { get; set; }

    [ForeignKey("StyleId")]
    public Style Style { get; set; }
  }
}
```

Для создания отношения с помощью свойств, которые согласованы с остальными частями модели данных, требуются два атрибута. Атрибут `Column` сообщает инфраструктуре Entity Framework Core о том, что свойство `StyleId` должно отображаться на столбец `ColorId`, в то время как атрибут `ForeignKey` указывает, что свойство `StyleId` является свойством внешнего ключа для навигационного свойства `Style`. Вместе эти атрибуты позволяют свойствам, имена которых согласуются с остальными частями модели данных, выражать отношение без необходимости в раскрытии деталей внутренней структуры БД.

Свойство, добавленное к классу `Styles` в листинге 18.14, не требует атрибута, поскольку оно следует обычным соглашениям, касающимся отношений, и Entity Framework Core распознает его как обратное навигационное свойство для класса `Shoe`. Однако если применить согласованное имя для такого типа свойства невозможно, тогда можете воспользоваться атрибутом `InverseProperty` и сообщить инфраструктуре Entity Framework Core, с каким классом необходимо связать это свойство (листинг 18.15).

Листинг 18.15. Идентификация обратного навигационного свойства в файле `Styles.cs` из папки `Models/Manual`

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Collections.Generic;

namespace ExistingDb.Models.Manual {

  [Table("Colors")]
  public class Style {

    [Key]
    [Column("Id")]
    public long UniqueIdent { get; set; }

    [Column("Name")]
    public string StyleName { get; set; }

    public string MainColor { get; set; }
    public string HighlightColor { get; set; }

    [InverseProperty(nameof(Shoe.Style))]
    public IEnumerable<Shoe> Products { get; set; }
  }
}
```

В аргументе атрибуту `InverseProperty` передается имя другого свойства в отношении, которое может быть указано как строка или с помощью функции `nameof()` во избежание опечаток. В листинге 18.15 атрибут `InverseProperty` позволил изменить имя обратного навигационного свойства на `Products`.

В табл. 18.6 описаны атрибуты, которые применялись в текущем разделе для переопределения соглашений, касающихся отношений. (Атрибут `Column` был описан в табл. 18.3.)

Таблица 18.6. Атрибуты, которые использовались для переопределения соглашений, касающихся отношений

Имя	Описание
`ForeignKey(property)`	Применяется для идентификации свойства внешнего ключа, связанного с навигационным свойством
`InverseProperty(name)`	Используется для указания имени свойства на другой стороне отношения

Переопределение соглашений, касающихся отношений, с использованием Fluent API

Если вы отдаете предпочтение интерфейсу Fluent API, то можете посредством метода `Entity<T>()` выбрать сущностный класс и затем вызвать метод `Property()`, который позволяет выбирать и конфигурировать индивидуальные свойства. Чтобы посмотреть, как применять Fluent API для описания отношений, добавьте в класс `Shoe` свойство внешнего ключа и навигационное свойство, создав отношение с классом `ShoeWidth`, которое представляет данные из таблицы БД по имени `Fittings` (листинг 18.16).

Листинг 18.16. Определение свойств в файле Shoe.cs из папки Models/Manual

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ExistingDb.Models.Manual {
  public class Shoe {
    public long Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    [Column("ColorId")]
    public long StyleId { get; set; }
    [ForeignKey("StyleId")]
    public Style Style { get; set; }
    public long WidthId { get; set; }
    public ShoeWidth Width { get; set; }
  }
}
```

Для укомплектования отношения добавьте в класс ShoeWidth обратное навигационное свойство (листинг 18.17).

Листинг 18.17. Добавление обратного навигационного свойства в файле ShoeWidth.cs из папки Models/Manual

```
using System.Collections.Generic;

namespace ExistingDb.Models.Manual {
  public class ShoeWidth {
    public long UniqueIdent { get; set; }
    public string WidthName { get; set; }
    public IEnumerable<Shoe> Products { get; set; }
  }
}
```

Добавленные свойства не следуют соглашениям, принятым для отношений, и не декорированы атрибутами, а потому инфраструктура Entity Framework Core не в состоянии определить их предназначение. Чтобы описать роль этих свойств в модели данных, добавьте в метод OnModelCreating() класса ManualContext операторы, выделенные полужирным в листинге 18.18.

Листинг 18.18. Переопределение соглашений в файле ManualContext.cs из папки Models/Manual

```
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Models.Manual {
  public class ManualContext : DbContext {
    public ManualContext(DbContextOptions<ManualContext> options)
      : base(options) { }
    public DbSet<Shoe> Shoes { get; set; }
    public DbSet<Style> ShoeStyles { get; set; }
    public DbSet<ShoeWidth> ShoeWidths { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<ShoeWidth>().ToTable("Fittings");
      modelBuilder.Entity<ShoeWidth>().HasKey( t => t.UniqueIdent);
```

```
        modelBuilder.Entity<ShoeWidth>()
          .Property(t => t.UniqueIdent)
          .HasColumnName("Id");
        modelBuilder.Entity<ShoeWidth>()
          .Property(t => t.WidthName)
          .HasColumnName("Name");
        modelBuilder.Entity<Shoe>()
          .Property(s => s.WidthId).HasColumnName("FittingId");
        modelBuilder.Entity<Shoe>()
          .HasOne(s => s.Width).WithMany(w => w.Products)
          .HasForeignKey(s => s.WidthId).IsRequired(true);
    }
  }
}
```

Для конфигурирования свойств, определенных в листингах 18.17 и 18.18, с целью представления отношения между таблицами `Shoes` и `Fittings` потребовались два оператора. В первом операторе используются те же методы из предыдущего раздела, которые сообщают инфраструктуре Entity Framework Core о том, что значения для свойства `Shoe.WidthId` должны читаться из столбца `FittingId`. Во втором операторе применяются методы, которые Fluent API предоставляет специально для описания отношений; они перечислены в табл. 18.7.

Таблица 18.7. Методы `EntityBuilder<T>` для описания отношений

Имя	Описание
`HasOne(property)`	Используется для того, чтобы начать описание отношения, где выбранный сущностный класс имеет отношение с одним объектом другого типа. Аргумент выбирает навигационное свойство либо по имени, либо с помощью лямбда-выражения
`HasMany(property)`	Применяется для того, чтобы начать описание отношения, где выбранный сущностный класс имеет отношение со многими объектами другого типа. Аргумент выбирает навигационное свойство либо по имени, либо с помощью лямбда-выражения

Вы можете начать с одного из методов, показанных в табл. 18.7, и описать другую сторону отношения, используя один из методов, которые перечислены в табл. 18.8; они указывают инфраструктуре Entity Framework Core вид отношения — "один к одному" или "один ко многим".

Таблица 18.8. Методы Fluent API, завершающие описание отношения

Имя	Описание
`WithMany(property)`	Применяется для выбора обратного навигационного свойства в отношении "один ко многим"
`WithOne(property)`	Используется для выбора обратного навигационного свойства в отношении "один к одному"

После выбора навигационных свойств для обеих сторон отношения можно сконфигурировать отношение, добавив в цепочку вызов одного из методов, которые представлены в табл. 18.9.

Таблица 18.9. Методы Fluent API для конфигурирования отношений

Имя	Описание
`HasForeignKey(property)`	Применяется для выбора свойства внешнего ключа, связанного с отношением
`IsRequired(required)`	Используется для указания, каким должно быть отношение — обязательным или необязательным

Комбинация методов в листинге 18.18 сообщает инфраструктуре Entity Framework Core о том, что класс `Shoe` является зависимой сущностью в обязательном отношении "один ко многим" с классом `ShoeWidth` и свойство `WidthId` выступает в качестве свойства внешнего ключа. В сочетании с оператором, который отображает свойство `WidthId` на столбец внешнего ключа, инфраструктура Entity Framework Core располагает всей необходимой информацией, чтобы понять отношение и способ отображения сущностных классов на таблицы БД.

Завершение модели данных

Чтобы завершить модель данных, осталось определить классы для представления таблиц `SalesCampaigns` и `Categories` и описать отношения с ними. Добавьте в папку `Models/Manual` файл класса по имени `SalesCampaign.cs` с определением из листинга 18.19.

Листинг 18.19. Содержимое файла `SalesCampaign.cs` из папки `Models/Manual`

```
using System;
using System.ComponentModel.DataAnnotations.Schema;

namespace ExistingDb.Models.Manual {
  [Table("SalesCampaigns")]
  public class SalesCampaign {
    public long Id { get; set; }
    public string Slogan { get; set; }
    public int? MaxDiscount { get; set; }
    public DateTime? LaunchDate { get; set; }
    public long ShoeId { get; set; }
    public Shoe Shoe { get; set; }
  }
}
```

Атрибут `Table` применяется для указания таблицы, которая будет хранить объекты `SalesCampaign`, позволяя поддерживать имена классов согласованными в рамках модели данных. Свойства, определенные в классе `SalesCampaign`, отображаются на столбцы в таблице БД напрямую за исключением свойства `Shoe`, которое является навигационным свойством для отношения "один к одному" с классом `Shoe`.

Таблица `Categories` имеет отношение "многие ко многим", т.е. понадобится определить сущностный и соединяющий классы. Для сущностного класса добавьте в папку `Models/Manual` файл по имени `Category.cs` и поместите в него определение из листинга 18.20.

Листинг 18.20. Содержимое файла `Category.cs` из папки `Models/Manual`

```
using System.Collections.Generic;

namespace ExistingDb.Models.Manual {
  public class Category {
    public long Id { get; set; }
    public string Name { get; set; }
    public IEnumerable<ShoeCategoryJunction> Shoes { get; set; }
  }
}
```

Для представления соединяющего класса, необходимого отношению "многие ко многим", добавьте в папку `Models/Manual` файл по имени `ShoeCategoryJunction.cs`, содержимое которого приведено в листинге 18.21.

Листинг 18.21. Содержимое файла `ShoeCategoryJunction.cs` из папки `Models/Manual`

```
namespace ExistingDb.Models.Manual {

  public class ShoeCategoryJunction {
    public long Id { get; set; }
    public long ShoeId { get; set; }
    public long CategoryId { get; set; }
    public Category Category { get; set; }
    public Shoe Shoe { get; set; }
  }
}
```

В классе `ShoeCategoryJunction` определены свойства, соответствующие столбцам в соединяющей таблице, а также навигационные свойства, использующие имена классов в модели данных, которые применяются вместо имен, принятых по соглашениям. Имя класса совпадает с именем соединяющей таблицы в БД. Поскольку соединяющий класс следует всем соглашениям, для конфигурирования этого класса либо его отношений какие-то атрибуты или операторы Fluent API не требуются.

Добавьте в класс `Shoe` навигационные свойства, которые требуются для класса `SalesCampaign` и соединяющего класса в отношении с классом `Category` из листинга 18.21, как показано в листинге 18.22.

Листинг 18.22. Добавление навигационных свойств в файле `Shoe.cs` из папки `Models/Manual`

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Collections.Generic;

namespace ExistingDb.Models.Manual {
  public class Shoe {
    public long Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    [Column("ColorId")]
    public long StyleId { get; set; }
    [ForeignKey("StyleId")]
    public Style Style { get; set; }
```

```
    public long WidthId { get; set; }
    public ShoeWidth Width { get; set; }
    public SalesCampaign Campaign { get; set; }
    public IEnumerable<ShoeCategoryJunction> Categories { get; set; }
  }
}
```

Чтобы обеспечить удобный доступ к объектам `Category`, добавьте в класс контекста свойство `DbSet<T>` (листинг 18.23).

Листинг 18.23. Конфигурирование отношения в файле `ManualContext.cs` из папки `Models/Manual`

```
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Models.Manual {

  public class ManualContext : DbContext {

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

    public DbSet<Shoe> Shoes { get; set; }

    public DbSet<Style> ShoeStyles { get; set; }

    public DbSet<ShoeWidth> ShoeWidths { get; set; }

    public DbSet<Category> Categories { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      // ...для краткости операторы не показаны...
    }
  }
}
```

Использование модели данных, созданной вручную

Модель данных завершена, и данные можно задействовать в части ASP.NET Core MVC приложения, позволив инфраструктуре Entity Framework Core обрабатывать их отображение на БД, когда соглашения не соблюдались. В последующих разделах будет демонстрироваться процесс запрашивания и обновления данных.

Запрашивание данных в модели данных, созданной вручную

Расширьте запрос в действии `Index` контроллера `Manual` с целью включения всех доступных данных и добавьте свойства `ViewBag` для каждого типа данных, чтобы запрашивать все отношения в модели (листинг 18.24).

Листинг 18.24. Запрашивание дополнительных данных в файле `ManualController.cs` из папки `Models/Manual`

```
using ExistingDb.Models.Manual;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace ExistingDb.Controllers {
```

```
  public class ManualController : Controller {
    private ManualContext context;

    public ManualController(ManualContext ctx) => context = ctx;

    public IActionResult Index() {
      ViewBag.Styles = context.ShoeStyles.Include(s => s.Products);
      ViewBag.Widths = context.ShoeWidths.Include(s => s.Products);
      ViewBag.Categories = context.Categories
        .Include(c => c.Shoes).ThenInclude(j => j.Shoe);
      return View(context.Shoes.Include(s => s.Style)
        .Include(s => s.Width).Include(s => s.Categories)
          .ThenInclude(j => j.Category));
    }
  }
}
```

Для отображения данных пользователю добавьте в представление `Index`, применяемое методом действия, элементы из листинга 18.25, которые выделены полужирным.

Листинг 18.25. Отображение данных в файле `Index.cshtml` из папки `Views/Manual`

```
@using ExistingDb.Models.Manual
@model IEnumerable<Shoe>
@{
  ViewData["Title"] = "Manual Data Model";
  Layout = "_Layout";
}

<div class="container-fluid">
  <h4 class="bg-primary p-3 text-white">Zoom Shoes</h4>
  <table class="table table-striped table-sm">
    <tr>
      <th>Id</th><th>Name</th><th>Price</th><th>Styles</th>
      <th>Widths</th><th>Categories</th><th></th>
    </tr>
    @foreach (Shoe s in Model) {
      <tr>
        <td>@s.Id</td>
        <td>@s.Name</td>
        <td>$@s.Price.ToString("F2")</td>
        <td class="table-primary">@s.Width?.WidthName</td>
        <td class="table-secondary">@s.Style?.StyleName</td>
        <td class="table-success">
          @string.Join(", ", s.Categories.Select(c => c.Category.Name))
        </td>
        <td class="text-center">
          <a asp-action="Edit" asp-route-id="@s.Id"
             class="btn btn-sm btn-primary">Edit</a>
        </td>
      </tr>
    }
  </table>
```

```
<div class="row">
  <div class="col">
    <h5 class="bg-primary p-2 text-white">Styles</h5>
    <table class="table table-striped table-sm">
      <tr><th>UniqueIdent</th><th>Name</th><th>Products</th></tr>
      @foreach (Style s in ViewBag.Styles) {
        <tr>
          <td>@s.UniqueIdent</td>
          <td>@s.StyleName</td>
          <td class="table-secondary">
            @String.Join(", ", s.Products.Select(p => p.Name))
          </td>
        </tr>
      }
    </table>
  </div>
  <div class="col">
    <h5 class="bg-primary p-2 text-white">Widths</h5>
    <table class="table table-striped table-sm">
      <tr><th>UniqueIdent</th><th>Name</th><td>Products</td></tr>
      @foreach (ShoeWidth s in ViewBag.Widths) {
        <tr>
          <td>@s.UniqueIdent</td>
          <td>@s.WidthName</td>
          <td class="table-primary">
            @String.Join(", ", s.Products.Select(p => p.Name))
          </td>
        </tr>
      }
    </table>
  </div>
  <div class="col">
    <h5 class="bg-primary p-2 text-white">Categories</h5>
    <table class="table table-striped table-sm">
      <tr><th>Id</th><th>Name</th><th>Products</th></tr>
      @foreach (Category c in ViewBag.Categories) {
        <tr>
          <td>@c.Id</td>
          <td>@c.Name</td>
          <td class="table-success">
            @String.Join(", ", c.Shoes.Select(j => j.Shoe.Name))
          </td>
        </tr>
      }
    </table>
  </div>
</div>
</div>
```

Запустите приложение и перейдите по ссылке `http://localhost:5000/manual`; появится новое содержимое (рис. 18.4).

Рис. 18.4. Использование модели данных, созданной вручную

Результатом добавлений в листинге 18.25 оказывается ряд таблиц, отображающих данные в БД. Результат демонстрирует, что работа с созданной вручную моделью данных производится в точности так же, как с моделью данных, сгенерированной процессом формирования шаблонов или применяемой в качестве основы для миграции. Инфраструктура Entity Framework Core автоматически заботится о любых отклонениях от соглашений, что можно заметить, исследуя вывод из приложения. Вот один из запросов, используемых для получения отображенных на рис. 18.4 данных; вы можете видеть, каким образом имена классов и свойств, не соблюдающих соглашения, транслируются в таблицы и столбцы в БД:

```
...
SELECT [c.Shoes].[Id], [c.Shoes].[CategoryId],
  [c.Shoes].[ShoeId], [s.Shoe].[Id],
  [s.Shoe].[Name], [s.Shoe].[Price],
  [s.Shoe].[ColorId], [s.Shoe].[FittingId]
FROM [ShoeCategoryJunction] AS [c.Shoes]
INNER JOIN [Shoes] AS [s.Shoe] ON [c.Shoes].[ShoeId] = [s.Shoe].[Id]
INNER JOIN (
  SELECT [c0].[Id]
  FROM [Categories] AS [c0]
) AS [t] ON [c.Shoes].[CategoryId] = [t].[Id]
ORDER BY [t].[Id]
...
```

Внимание! Создание ручной модели данных приводит к более естественной практике разработки в части ASP.NET Core MVC приложения, но оно действует только в случае точного моделирования БД. Инфраструктура Entity Framework Core не проверяет, точно ли созданная модель описывает БД, и любые проблемы проявятся лишь при попытке запрашивания или обновления БД.

Обновление данных в модели данных, созданной вручную

В листинге 18.25 к представлению был добавлен якорный элемент, нацеленный на действие по имени Edit. Это действие необходимо реализовать, акцентируя внимание на том, что работа с моделью данных, созданной вручную, ничем не отличается от работы с моделью данных, сгенерированной процессом формирования шаблонов или применяемой для создания миграции. Добавьте в класс ManualController методы действий, которые позволят пользователю начать процесс редактирования и обновить существующий объект Shoe (листинг 18.26).

Листинг 18.26. Добавление методов действий в файле ManualController.cs из папки Controllers

```
using ExistingDb.Models.Manual;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Collections.Generic;

namespace ExistingDb.Controllers {

  public class ManualController : Controller {
    private ManualContext context;
    public ManualController(ManualContext ctx) => context = ctx;
    public IActionResult Index() {
      ViewBag.Styles = context.ShoeStyles.Include(s => s.Products);
      ViewBag.Widths = context.ShoeWidths.Include(s => s.Products);
      ViewBag.Categories = context.Categories
        .Include(c => c.Shoes).ThenInclude(j => j.Shoe);
      return View(context.Shoes.Include(s => s.Style)
        .Include(s => s.Width).Include(s => s.Categories)
          .ThenInclude(j => j.Category));
    }
    public IActionResult Edit(long id) {
      ViewBag.Styles = context.ShoeStyles;
      ViewBag.Widths = context.ShoeWidths;
      ViewBag.Categories = context.Categories;
      return View(context.Shoes.Include(s => s.Style)
        .Include(s => s.Campaign)
        .Include(s => s.Width).Include(s => s.Categories)
          .ThenInclude(j => j.Category).First(s => s.Id == id));
    }
    [HttpPost]
    public IActionResult Update(Shoe shoe, long[] newCategoryIds,
        ShoeCategoryJunction[] oldJunctions) {
      IEnumerable<ShoeCategoryJunction> unchangedJunctions
        = oldJunctions.Where(j => newCategoryIds.Contains(j.CategoryId));
      context.Set<ShoeCategoryJunction>()
        .RemoveRange(oldJunctions.Except(unchangedJunctions));
      shoe.Categories = newCategoryIds.Except(unchangedJunctions
        .Select(j => j.CategoryId))
        .Select(id => new ShoeCategoryJunction {
          ShoeId = shoe.Id, CategoryId = id
      }).ToList();
```

```
            context.Shoes.Update(shoe);
            context.SaveChanges();
            return RedirectToAction(nameof(Index));
        }
    }
}
```

Метод `Edit()` запрашивает у БД одиночный объект и все связанные с ним данные, чтобы можно было отобразить текущие детали для редактирования. Дополнительные данные передаются представлению посредством `ViewBag`, и пользователь получает целый диапазон возможностей для изменения связанных данных.

Метод `Update()` принимает объект `Shoe`, который создается связывателем моделей ASP.NET Core MVC из данных формы в HTTP-запросе и используется для обновления БД. Вдобавок метод `Update()` получает массив, содержащий значения первичного ключа объектов `Category`, с которыми необходимо связать объект `Shoe`. Кроме того, метод `Update()` принимает массив соединяющих объектов, включенных в HTTP-запрос, чтобы не пришлось запрашивать БД с целью выяснения, какие соединяющие объекты должны быть удалены.

В главе 16 было показано, как обновлять отношение "многие ко многим" отдельно от объектов в этом отношении, для чего у БД запрашивался объект на одной стороне отношения вместе со связанными данными, что снабжало инфраструктуру Entity Framework Core достаточной информацией для обновления соединяющей таблицы.

В текущей главе такой прием не подходит, поскольку здесь нужно обновить объект `Shoe` и отношение "многие ко многим" одновременно. Дело в том, что если запросить у БД объект `Shoe`, то в случае применения метода `Update()` с объектом `Shoe`, созданным связывателем моделей MVC, средство отслеживания изменений Entity Framework Core сгенерирует исключение.

Чтобы выполнить обновление, необходимо удалить все устаревшие отношения, сравнивая выбранные категории с первоначально связанными категориями и удаляя ненужные путем передачи объектов методу `RemoveRange()`:

```
...
IEnumerable<ShoeCategoryJunction> unchangedJunctions
  = oldJunctions.Where(j => newCategoryIds.Contains(j.CategoryId));

context.Set<ShoeCategoryJunction>()
  .RemoveRange(oldJunctions.Except(unchangedJunctions));
...
```

Для соединяющих данных свойство контекста не определялось, поэтому операция удаления выполняется за счет получения объекта `DbSet<T>` с использованием метода `Set<T>()`. Во избежание создания дублированных отношений первоначальный набор соединяющих объектов сравнивается с категориями, которые выбрал пользователь, гарантируя создание только новых объектов для заполнения пробелов:

```
...
shoe.Categories =
 newCategoryIds.Except(unchangedJunctions.Select(j => j.CategoryId))
  .Select(id => new ShoeCategoryJunction {
    ShoeId = shoe.Id, CategoryId = id
}).ToList();
...
```

Нежелательные отношения удаляются, а новые создаются, оставляя в БД те отношения, которые не изменялись. Такой подход позволяет выполнить обновление с применением объектов, созданных связывателем моделей ASP.NET Core MVC.

Создание частичных представлений

Чтобы завершить пример, для всех объектов понадобится создать частичные представления, которые дадут пользователю возможность редактировать объекты. Добавьте в папку `Views/Manual` файл по имени `EditShoe.cshtml` с содержимым, показанным в листинге 18.27.

Листинг 18.27. Содержимое файла `EditShoe.cshtml` из папки `Views/Manual`

```
@using ExistingDb.Models.Manual
@model Shoe

<input type="hidden" asp-for="Id" />
<h4>Product Details</h4>
<div class="p-1 m-1">
  <div class="form-row">
    <div class="form-group col">
      <label asp-for="Name" class="form-control-label"></label>
      <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group col">
      <label asp-for="Price" class="form-control-label"></label>
      <input asp-for="Price" class="form-control" />
    </div>
  </div>
</div>
```

Каждое частичное представление содержит только те элементы, которые требуются для редактирования одного аспекта данных, отображаемых пользователю, и в `EditShoe.cshtml` имеются два элемента `input`, позволяющие редактировать свойства `Name` и `Price` объекта `Shoe`. Далее добавьте в папку `Views/Manual` файл по имени `EditStyle.cshtml` с содержимым из листинга 18.28.

Листинг 18.28. Содержимое файла `EditStyle.cshtml` из папки `Views/Manual`

```
@using ExistingDb.Models.Manual
@model Shoe

<label><strong>Style:</strong></label>
<select asp-for="StyleId" class="form-control">
  @foreach (Style s in ViewBag.Styles) {
    if (s.UniqueIdent == Model.StyleId) {
      <option value="@s.UniqueIdent" selected>@s.StyleName</option>
    } else {
      <option value="@s.UniqueIdent">@s.StyleName</option>
    }
  }
</select>
```

Частичное представление `EditStyle` отображает элемент `select`, предназначенный для выбора объекта `Style`, с которым связан редактируемый объект `Shoe`. Аналогичный подход используется для объекта `ShoeWidth`, поэтому создайте в папке

Views/Manual файл по имени EditWidth.cshtml и приведите его содержимое в соответствие с листингом 18.29.

Листинг 18.29. Содержимое файла EditWidth.cshtml из папки Views/Manual

```
@using ExistingDb.Models.Manual
@model Shoe

<label>Width:</label>
<select asp-for="WidthId" class="form-control">
  @foreach (ShoeWidth w in ViewBag.Widths) {
    if (w.UniqueIdent == Model.WidthId) {
      <option value="@w.UniqueIdent" selected>@w.WidthName</option>
    } else {
      <option value="@w.UniqueIdent">@w.WidthName</option>
    }
  }
</select>
```

Из-за наличия отношения "один к одному" с классом SalesCampaign пользователю необходимо отобразить элементы input для изменения значений свойств. Добавьте в папку Views/Manual файл по имени EditCampaign.cshtml с содержимым из листинга 18.30.

Листинг 18.30. Содержимое файла EditCampaign.cshtml из папки Views/Manual

```
@using ExistingDb.Models.Manual
@model Shoe

<div class="form-row">
  <input type="hidden" asp-for="Campaign.Id" />
  <label><strong>Sales Campaign:</strong></label>
</div>
<div class="form-row">
  <div class="form-group col">
    <label asp-for="Campaign.Slogan" class="form-control-label"></label>
    <input asp-for="Campaign.Slogan" class="form-control" />
  </div>
</div>
<div class="form-row">
  <div class="form-group col">
    <label class="form-control-label">Max Discount:</label>
    <input asp-for="Campaign.MaxDiscount" class="form-control" />
  </div>
  <div class="form-group col">
    <label class="form-control-label">Launch Date:</label>
    <input type="date" asp-for="Campaign.LaunchDate" class="form-control" />
  </div>
</div>
```

Последнее частичное представление позволит пользователю выбирать объекты Category, с которыми должен быть связан редактируемый объект Shoe. Оно несколько сложнее, т.к. включает скрытые элементы, которые связыватель моделей ASP.NET Core MVC будет применять для создания соединяющих объектов, используемых

вместо того, чтобы запрашивать у БД текущий набор отношений в методе действия Update(). Добавьте в папку Views/Manual файл по имени EditCategory.cshtml, содержимое которого приведено в листинге 18.31.

Листинг 18.31. Содержимое файла EditCategory.cshtml из папки Views/Manual

```
@using ExistingDb.Models.Manual
@model Shoe

@{ int index = 0; }

@foreach (var junc in Model.Categories) {
  <input type="hidden" name="oldJunctions[@index].Id" value="@junc.Id" />
  <input type="hidden" name="oldJunctions[@index].CategoryId"
       value="@junc.CategoryId" />
  index++;
}

@foreach (Category c in ViewBag.Categories) {
  <div class="form-group col">
    <label class="form-check-label">
      @if (c.Shoes?.Any(s => s.ShoeId == Model.Id) == true) {
        <input type="checkbox" name="newCategoryIds" value="@c.Id"
              checked class="form-check-input" />
      } else {
        <input type="checkbox" name="newCategoryIds" value="@c.Id"
              class="form-check-input" />
      }
      @c.Name
    </label>
  </div>
}
```

С виду неуклюжий цикл foreach со счетчиком index применяется для генерирования данных, которые будут корректно обрабатываться связывателем моделей ASP.NET Core MVC. Остальные элементы создают серию меток и флажков, предназначенных для выбора категорий, к которым относится объект Shoe.

Создание представления редактора

Осталось лишь создать представление, которое объединит частичные представления, чтобы снабдить пользователя единым редактором. Добавьте в папку Views/Manual файл по имени Edit.cshtml с содержимым, показанным в листинге 18.32.

Листинг 18.32. Содержимое файла Edit.cshtml из папки Views/Manual

```
@using ExistingDb.Models.Manual
@model Shoe
@{
  ViewData["Title"] = "Manual Data Model";
  Layout = "_Layout";
}

<form asp-action="Update" method="post">
  @Html.Partial("EditShoe", Model)
```

```
  <div class="p-1 m-1">
    <div class="form-row">
      <div class="form-group col">@Html.Partial("EditStyle", Model)</div>
      <div class="form-group col">@Html.Partial("EditWidth", Model)</div>
    </div>
    @Html.Partial("EditCampaign", Model)
    <div class="form-row">
      <label><strong>Categories:</strong></label>
    </div>
    <div class="form-row">@Html.Partial("EditCategory", Model)</div>
  </div>
  <div class="text-center m-1">
    <button type="submit" class="btn btn-primary">Save</button>
    <a asp-action="Index" class="btn btn-secondary">Cancel</a>
  </div>
</form>
```

Чтобы убедиться в работоспособности кода и представлений для обновления данных в модели данных, созданной вручную, запустите приложение, перейдите по ссылке `http://localhost:5000/manual` и щелкните на кнопке Edit (Редактировать) рядом с товаром `All Terrain Monster`. В раскрывающемся списке Width (Ширина) выберите вариант Big Foot (Большая нога) и снимите отметку с флажка Trail (Тропа) в разделе Categories (Категории). Щелкнув на кнопке Save (Сохранить), вы увидите, что внесенные изменения отразились в общем представлении (рис. 18.5).

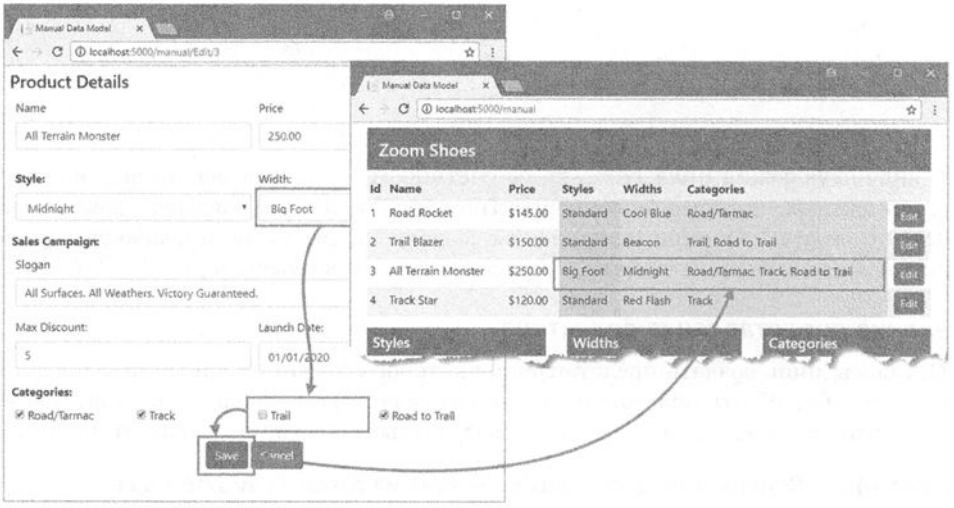

Рис. 18.5. Редактирование для модели данных, созданной вручную

Резюме

В главе демонстрировался процесс ручного моделирования БД. Он требует особого внимания, но может в итоге дать модель, которую легче использовать в части ASP.NET Core MVC приложения по сравнению с моделью, полученной в результате выполнения процесса формирования шаблонов, который рассматривался в главе 17. В третьей части книги будут описаны расширенные возможности Entity Framework Core.

ЧАСТЬ III

Расширенные возможности инфраструктуры Entity Framework Core 2

В третьей части книги рассматриваются расширенные возможности, предлагаемые инфраструктурой Entity Framework Core. Они требуются далеко не в каждом проекте, но могут быть неоценимыми, когда необходимо выйти за рамки стандартных средств, описанных во второй части.

ГЛАВА 19

Работа с ключами

Выбор наиболее подходящих ключей для имеющихся данных устанавливает фундамент для остальных частей модели данных. В предшествующих главах вы ознакомились с ролью, которую играют ключи в уникальной идентификации объектов, узнали о соглашениях, применяемых инфраструктурой Entity Framework Core для выбора свойств с целью использования в качестве ключей, и научились переопределять принятые соглашения. В этой главе будут описаны расширенные функции ключей, поддерживаемые Entity Framework Core. Подобно многим средствам, обсуждаемым в третьей части книги, потребность в расширенных функциях ключей вряд ли будет возникать в каждом проекте, в котором применяется инфраструктура Entity Framework Core. Однако возможность взятия под свой контроль того, как используются ключи, может быть важной в необычных ситуациях, когда стандартные функции не способны обеспечить функциональность, требуемую в приложении.

Инфраструктура Entity Framework Core не в состоянии генерировать миграции, которые вносят значимые изменения в ключи, а потому большинство рассматриваемых в главе примеров требуют переустановки БД или удаления более ранних миграций. Это подчеркивает важность выбора стратегии ключей как можно раньше, чтобы избежать необходимости вносить сложные изменения в БД, которые могут привести к потере данных. В табл. 19.1 приведены сведения, позволяющие поместить расширенные функции ключей в контекст.

Таблица 19.1. Помещение расширенных функций ключей в контекст

Вопрос	Ответ
Что это такое?	Расширенные функции ключей позволяют изменять способ создания и использования ключей
Чем они полезны?	Не все приложения могут работать со стандартными функциями первичного ключа, особенно при работе с существующей БД
Как они используются?	Расширенные функции ключей применяются с помощью операторов Fluent API в классе контекста БД
Существуют ли какие-то скрытые ловушки или ограничения?	Расширенные функции ключей требуют тщательного обдумывания, потому что очень легко выбрать свойства, значения которых не будут идентифицировать объект уникальным образом
Существуют ли альтернативы?	Расширенные функции ключей не являются обязательными, и вы можете использовать стандартные функции, описанные во второй части

На заметку! Многие расширенные функции могут применяться только с использованием Fluent API и не имеют соответствующих атрибутов. О доступности атрибутов упоминается особо, но основное внимание в этой и в остальных главах третьей части книги сосредоточено главным образом на работе с Fluent API.

В табл. 19.2 приведена сводка по главе.

Таблица 19.2. Сводка по главе

Задача	Решение	Листинг
Изменение способа, которым генерируются значения первичного ключа	Применяйте методы генерации ключей из Fluent API	19.14–19.16
Использование естественных ключей	Применяйте метод `IsUnique()` для гарантирования того, что область значений в БД не дублируется	19.17, 19.18, 19.25–19.27
Использование дополнительных свойств для идентификации объектов	Создайте альтернативный ключ	19.19–19.24
Применение множества свойств для идентификации объектов	Создайте составной ключ	19.28–19.32

Подготовительные шаги

В главе создается новый проект, который позволит продемонстрировать расширенные функции, поддерживаемые инфраструктурой Entity Framework Core. Запустите Visual Studio, выберите в меню File (Файл) пункт New⇨Project (Создать⇨Проект) и укажите шаблон проекта ASP.NET Core Web Application (Веб-приложение ASP.NET Core), чтобы создать проект по имени `AdvancedApp` (рис. 19.1).

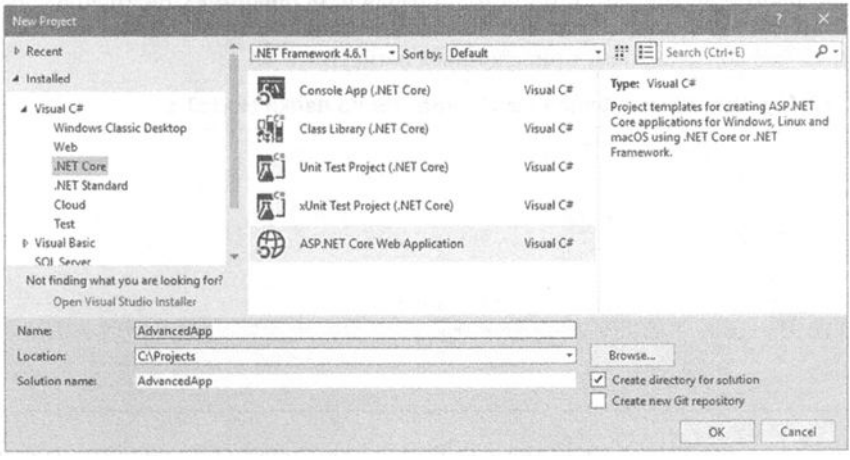

Рис. 19.1. Создание нового приложения

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Щелкните на кнопке OK для перехода в новое диалоговое окно. Удостоверьтесь, что в списке в левой верхней части окна выбран вариант ASP.NET Core 2.0, и щелкните на шаблоне Empty (Пустой), как показано на рис. 19.2. Щелкните на кнопке OK, чтобы закрыть диалоговое окно и создать проект.

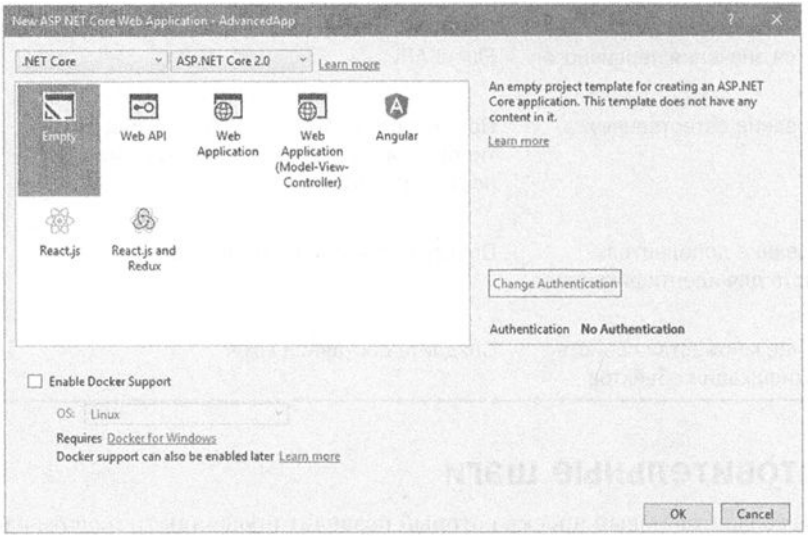

Рис. 19.2. Настройка проекта

Создание модели данных

Модель данных для настоящей главы будет представлять служащих в упрощенной БД кадров ради некоторого отличия от примеров, основанных на товарах, из предшествующих глав. Создайте папку `Models` и добавьте в нее файл класса по имени `Employee.cs` с содержимым, приведенным в листинге 19.1.

Листинг 19.1. Содержимое файла `Employee.cs` из папки `Models`

```
namespace AdvancedApp.Models {

  public class Employee {
    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }
    public decimal Salary { get; set; }
  }
}
```

Класс `Employee` имеет свойства для номера карточки социального страхования (`SSN`), имени и фамилии (`FirstName` и `FamilyName`), а также оклада (`Salary`) особы.

Реальная БД кадров потребовала бы дополнительных деталей, но для начала вполне достаточно той информации, что есть. Чтобы создать класс контекста БД, добавьте в папку Models файл класса по имени AdvancedContext.cs с определением, показанным в листинге 19.2.

Листинг 19.2. Содержимое файла AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {
  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) { }
    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
    }
  }
}
```

В классе контекста определено свойство DbSet<T> для обеспечения удобного доступа к объектам Employee в БД и переопределен метод OnModelCreating(), так что с помощью интерфейса Fluent API можно будет конфигурировать БД. В данный момент операторы конфигурации не требуются, поэтому для класса Employee можно благополучно использовать стандартные соглашения.

Создание контроллера и представлений

Для построения части ASP.NET Core MVC приложения начните с создания папки Controllers и добавления в нее файла класса по имени HomeController.cs с содержимым из листинга 19.3. Контроллер определяет действие Index, которое отображает данные пользователю, и действие Update, создающее и обновляющее объекты. В главе 20 будет введена возможность мягкого удаления объектов (т.е. они скрываются от пользователя, но по-прежнему находятся в БД), а в главе 22 рассмотрено действительное удаление объектов.

Листинг 19.3. Содержимое файла HomeController.cs из папки Controllers

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

namespace AdvancedApp.Controllers {
  public class HomeController : Controller {
    private AdvancedContext context;
    public HomeController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Employees);
    }
    public IActionResult Edit(long id) {
      return View(id == default(long)
        ? new Employee() : context.Employees.Find(id));
    }

    [HttpPost]
    public IActionResult Update(Employee employee) {
```

```
      if (employee.Id == default(long)) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы снабдить контроллер его представлением, создайте папку `Views/Home` и добавьте в нее файл по имени `Index.cshtml` с содержимым, приведенным в листинге 19.4. Представление `Index` отображает таблицу с деталями объектов `Employee`, прочитанных из БД, наряду с кнопками, которые позволят создавать и модифицировать объекты.

Листинг 19.4. Содержимое файла `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>Key</th>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="7" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.Id</td>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td class="text-right">
          <a asp-action="Edit" asp-route-id="@e.Id"
             class="btn btn-sm btn-primary">Edit</a>
        </td>
      </tr>
    }
  </tbody>
</table>
```

```
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Чтобы дать пользователю возможность создавать или редактировать объект Employee, добавьте в папку Views/Home файл по имени Edit.cshtml, содержимое которого показано в листинге 19.5.

Листинг 19.5. Содержимое файла Edit.cshtml из папки Views/Home

```
@model Employee
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h4 class="bg-info p-2 text-center text-white">
  Create/Edit
</h4>
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <div class="form-group">
    <label class="form-control-label" asp-for="SSN"></label>
    <input class="form-control" asp-for="SSN" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FirstName"></label>
    <input class="form-control" asp-for="FirstName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FamilyName"></label>
    <input class="form-control" asp-for="FamilyName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="Salary"></label>
    <input class="form-control" asp-for="Salary" />
  </div>
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Save</button>
    <a class="btn btn-secondary" asp-action="Index">Cancel</a>
  </div>
</form>
```

Чтобы обеспечить компоновку для представления, создайте папку Views/Shared и поместите в нее файл по имени _Layout.cshtml с содержимым из листинга 19.6.

Листинг 19.6. 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" />
```

```
  <style>
    .placeholder { visibility: collapse }
    .placeholder:only-child { visibility: visible }
  </style>
</head>
<body>
  <div class="p-2">
    @RenderBody()
  </div>
</body>
</html>
```

Компоновка включает ссылку на файл, который содержит CSS-стили Bootstrap и ряд специальных CSS-стилей, которые будут отображать элемент, добавленный в класс `placeholder`, в представлении `Index.cshtml`, когда данные для отображения отсутствуют.

Чтобы включить вспомогательные функции дескрипторов и импортировать пакет, который содержит классы модели для применения в представлениях, добавьте в папку `Views` файл по имени `_ViewImports.cshtml`, с содержимым из листинга 19.7.

Листинг 19.7. Содержимое файла `_ViewImports.cshtml` из папки `Views`

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

Конфигурирование приложения

Для установки пакета NuGet с инструментами командной строки Entity Framework Core щелкните правой кнопкой мыши на элементе проекта `AdvancedApp` в окне Solution Explorer, выберите в контекстном меню пункт Edit AdvancedApp.csproj (Редактировать `AdvancedApp.csproj`) и добавьте элемент, выделенный полужирным в листинге 19.8.

Листинг 19.8. Добавление пакета NuGet в файле `AdvancedApp.csproj` из папки `AdvancedApp`

```
<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>
```

Чтобы сконфигурировать детали, связанные с БД в примере приложения, воспользуйтесь шаблоном элемента ASP.NET Configuration File (Файл конфигурации ASP.NET) для добавления к проекту файла по имени `appsettings.json` и внесите в него изменения, показанные в листинге 19.9. В дополнение к строке подключения конфигурируется система ведения журналов, так что инфраструктура Entity Framework Core будет отображать подробности SQL-запросов и команд, которые она отправляет серверу баз данных.

Листинг 19.9. Содержимое файла `appsettings.json` из папки `AdvancedApp`

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

Для включения промежуточного ПО ASP.NET Core MVC и Entity Framework Core добавьте в класс `Startup` операторы конфигурации, приведенные в листинге 19.10.

Листинг 19.10. Конфигурирование промежуточного ПО в файле `Startup.cs` из папки `AdvancedApp`

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

namespace AdvancedApp {
  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<AdvancedContext>(options =>
        options.UseSqlServer(conString));
    }
```

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

С применением шаблона элемента JSON File (Файл JSON), находящегося в категории ASP.NET Core⇨Web⇨General (ASP.NET Core⇨Веб⇨Общие), создайте файл по имени `.bowerrc` с содержимым, показанным в листинге 19.11. (Важно обратить внимание на имя файла: оно начинается с точки, содержит две буквы `r` и не имеет расширения.)

Листинг 19.11. Содержимое файла `.bowerrc` из папки `AdvancedApp`

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

Снова воспользуйтесь шаблоном элемента JSON File и создайте файл по имени `bower.json` с содержимым из листинга 19.12.

Листинг 19.12. Содержимое файла `bower.json` из папки `AdvancedApp`

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

После сохранения изменений в файле среда Visual Studio загрузит пакет Bootstrap и установит его в папку `wwwroot/lib`.

Для упрощения работы с приложением отредактируйте файл `Properties/launchSettings.json`, изменив два содержащихся в нем URL, чтобы они указывали на порт `5000` (листинг 19.13). Данный порт будет применяться в URL для демонстрации функциональных возможностей примера приложения.

Листинг 19.13. 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"
      }
    },
    "AdvancedApp": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Создание базы данных и тестирование приложения

Выполните в папке проекта AdvancedApp команды из листинга 19.14, чтобы создать и применить миграцию, которая настроит БД для хранения объектов Employee.

Листинг 19.14. Создание и применение миграции к БД

```
dotnet ef migrations add Initial
dotnet ef database update
```

В следующем разделе БД будет удалена и воссоздана, но перед продолжением важно удостовериться в работоспособности примера приложения. Запустите приложение, используя dotnet run, и перейдите по ссылке http://localhost:5000. В текущий момент данные в БД отсутствуют, так что вы увидите содержимое заполнителя. Щелкните на кнопке Create (Создать), заполните поля формы и щелкните на кнопке Save (Сохранить), чтобы сохранить новый объект Employee в БД (рис. 19.3).

Управление генерацией ключей

При работе с SQL Server доступны две стратегии генерации значений для первичных ключей, которые конфигурируются с применением методов, описанных в табл. 19.3.

Таблица 19.3. Методы генерации ключей

Имя	Описание
ForSqlServerUseIdentityColumns()	Выбирает для генерации ключей стратегию Identity
ForSqlServerUseSequenceHiLo()	Выбирает для генерации ключей стратегию Hi-Lo

На заметку! Не все серверы баз данных поддерживают указанные стратегии для генерации ключей. Для выяснения, какие стратегии доступны, обратитесь в документацию, сопровождающую пакет поставщика БД.

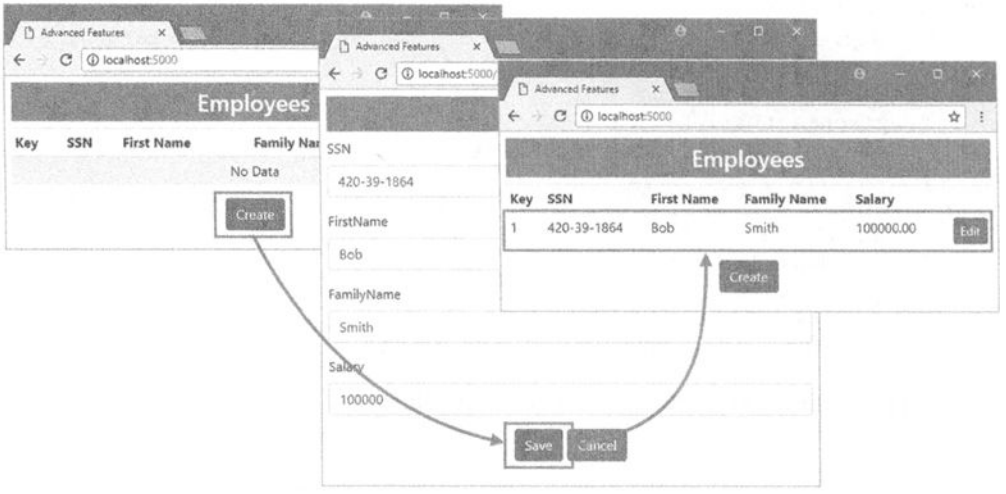

Рис. 19.3. Выполнение примера приложения

Стратегия Identity для генерации ключей

Стратегия Identity используется по умолчанию. При сохранении нового объекта инфраструктура Entity Framework Core рассчитывает, что уникальное значение первичного ключа создаст сервер баз данных. Таким образом, сохранение объекта требует две операции, которые можно увидеть, если щелкнуть на кнопке Create, заполнить форму, щелкнуть на кнопке Save и просмотреть журнальные сообщения, выданные приложением. Первая операция вставляет новые данные в БД:

```
...
INSERT INTO [Employees] ([FamilyName], [FirstName], [SSN], [Salary])
VALUES (@p0, @p1, @p2, @p3);
...
```

Инфраструктура Entity Framework Core не включает значение для свойства `Id`, поскольку ей известно, что значение будет присвоено сервером баз данных (и в действительности указание значения для `Id` в операции `UPDATE` приведет к ошибке). Вторая операция извлекает из БД первичный ключ, сгенерированный при вставке новых данных в таблицу:

```
...
SELECT [Id]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

Преимущество такого подхода — простота. Приложения, работающие с БД, не обязаны согласовывать действия друг с другом, чтобы избежать дублирования ключей, либо знать о том, как генерируются ключи. Недостаток подхода связан с потребностью в дополнительном запросе, получающем значение ключа.

Совет. Если вы не уверены, какой стратегии придерживаться, тогда выбирайте стратегию Identity, т.к. с ней легче всего работать и наименее вероятно столкнуться с проблемами.

Стратегия Hi-Lo для генерации ключей

Стратегия Hi-Lo представляет собой оптимизацию, которая позволяет инфраструктуре Entity Framework Core создавать значения первичных ключей вместо сервера баз данных, одновременно гарантируя уникальность этих значений. Чтобы взглянуть, как работает эта стратегия, потребуется провести небольшую работу, поскольку миграции Entity Framework Core не способны изменить стратегию генерации первичного ключа, которая была создана в более ранней миграции. Первый шаг предусматривает применение в классе контекста метода Fluent API из табл. 19.3 для выбора стратегии Hi-Lo, как показано в листинге 19.15.

Листинг 19.15. Выбор стратегии для генерации ключей в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) { }

    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .Property(e => e.Id).ForSqlServerUseSequenceHiLo();
    }
  }
}
```

Стратегия применяется путем выбора свойства и вызова метода `ForSqlServerUseSequenceHiLo()`, описанного в табл. 19.3. Для изменения стратегии генерации ключей понадобится удалить существующую миграцию и создать новую, чтобы модель данных создавалась в единственной миграции, и не требовалось вносить какие-то изменения в стратегию Identity. Выполните в папке проекта AdvancedApp команды из листинга 19.16 для удаления существующей миграции и создания ее замены.

Листинг 19.16. Переустановка миграции

```
dotnet ef migrations remove --force
dotnet ef migrations add HiLoStrategy
```

Просмотрев код метода `Up()` в файле `<отметка времени>_HiLoStrategy.cs` из папки `Migrations`, вы заметите, что была настроена новая последовательность по имени `EntityFrameworkHiLoSequence`:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.CreateSequence(
    name: "EntityFrameworkHiLoSequence",
    incrementBy: 10);

  migrationBuilder.CreateTable(
    name: "Employees",
```

```
        columns: table => new {
          Id = table.Column<long>(nullable: false),
          FamilyName = table.Column<string>(nullable: true),
          FirstName = table.Column<string>(nullable: true),
          SSN = table.Column<string>(nullable: true),
          Salary = table.Column<decimal>(nullable: false)
        },
        constraints: table => {
          table.PrimaryKey("PK_Employees", x => x.Id);
        });
    }
    ...
```

Новая последовательность будет использоваться для создания значений первичного ключа, как вскоре будет объяснено. Выполните в папке проекта `AdvancedApp` команды из листинга 19.17, чтобы удалить и воссоздать БД с применением новой миграции.

Листинг 19.17. Воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Использование стратегии Hi-Lo

В случае выбора стратегии Hi-Lo инфраструктура Entity Framework Core возлагает на себя ответственность за генерирование значений первичного ключа на основе начального значения, полученного от сервера баз данных. Когда приложение нуждается в сохранении объекта, инфраструктура Entity Framework Core получает очередное значение в последовательности `EntityFrameworkHiLoSequence` и трактует его как первое число в блоке из десяти значений первичного ключа, которые она может создать, не обращаясь к серверу баз данных и не согласовывая свои действия с другими приложениями. Например, если очередным значением последовательности является 100, тогда инфраструктуре Entity Framework Core известно, что она может создать объекты с первичными ключами 100, 101, 102 и т.д. вплоть до 109. После того как блок первичных ключей израсходован, из последовательности читается следующее значение. Каждое приложение (или экземпляр одного приложения) придерживается того же самого процесса для получения собственного блока ключей, гарантируя отсутствие дублированных значений ключей. Сервер баз данных обеспечивает для каждого запроса последовательности значений получение отличающегося результата и отсутствие дублированных блоков ключей.

Чтобы увидеть, как работает стратегия, запустите приложение, перейдите по ссылке `http://localhost:5000` и пройдите через процесс создания и сохранения нового объекта `Employee`. Очевидной разницы в поведении части ASP.NET Core MVC не будет, а изменение можно заметить, только исследуя журнальные сообщения Entity Framework Core.

При сохранении нового объекта инфраструктура Entity Framework Core получает очередное значение из последовательности:

```
...
SELECT NEXT VALUE FOR [EntityFrameworkHiLoSequence]
...
```

Значение последовательности представляет собой начало блока из десяти первичных ключей, которые Entity Framework Core может применять без необходимости в добавочных проверках. Это известно как "высокая" ("high") часть ключа, которая входит в состав названия стратегии Hi-Lo. "Низкая" ("low") часть порождается инкрементированием значения последовательности с целью генерации блока ключей, которая включается в операцию INSERT:

```
...
INSERT INTO [Employees] ([Id], [FamilyName], [FirstName], [SSN], [Salary])
VALUES (@p0, @p1, @p2, @p3, @p4);
...
```

В отличие от стратегии Identity инфраструктура Entity Framework Core не обязана запрашивать БД для выяснения значения первичного ключа. Каждый блок ключей разделяется между объектами контекста, созданными приложением, так что инфраструктуре Entity Framework Core придется лишь запросить очередное значение в последовательности после того, как она сохранит десять новых объектов.

Внимание! Не рассчитывайте на использование инфраструктурой Entity Framework Core какой-то специфической последовательности ключей. Реализация стратегии Hi-Lo может измениться или отличаться в случае перехода на другого поставщика БД.

Преимущество стратегии Hi-Lo в том, что она не требует запроса после каждой операции вставки для выяснения первичного ключа. Недостаток в том, что для работы стратегии Hi-Lo ей обязаны следовать все приложения, взаимодействующие с БД.

Исчерпание ключей Hi-Lo

Стратегия Hi-Lo может израсходовать возможный диапазон ключей, поскольку неиспользованные ключи в блоке будут "утрачены" при перезапуске приложения, поэтому для первичных ключей должен выбираться тип данных, обеспечивающий достаточную емкость. Чтобы посмотреть, как ключи остаются неиспользованными, остановите и перезапустите приложение, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create и сохраните в БД еще один объект `Employee`. Из БД будет прочитана новая последовательность значений и применена в качестве "высокого" компонента ключа, что даст результаты, показанные на рис. 19.4.

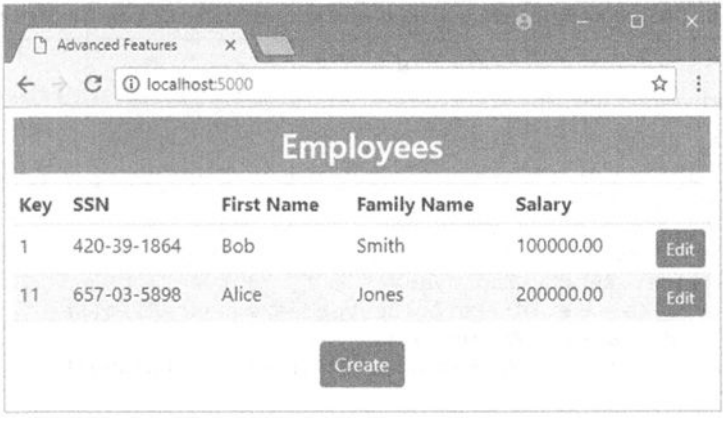

Рис. 19.4. Пропуск диапазона ключей при стратегии Hi-Lo

Ключи в диапазоне 2–10, находящиеся в блоке, который был получен предыдущим экземпляром приложения, использоваться не будут.

Работа с естественными ключами

Некоторые типы данных имеют собственные *естественные ключи*, а это означает наличие определенного аспекта, способного уникальным образом идентифицировать объект. В случае класса `Employee` свойство `SSN` может служить естественным ключом в странах, где можно рассчитывать на уникальность номеров карточек социального страхования.

Суррогатные ключи

Несмотря на то что данные `Employee` имеют естественный ключ, в класс `Employee` все равно добавлено выделенное свойство первичного ключа, которое применялось при демонстрации стратегий генерации ключей в предыдущем разделе. Такое свойство известно как *суррогатный ключ* и предназначено единственно для идентификации объекта; оно не имеет никаких отношений с остальными значениями, составляющими объект. Как будет показано в примерах далее в главе, вносить в модель данных изменения, воздействующие на первичный ключ, может быть нелегко, и использование суррогатного ключа помогает минимизировать влияние будущих изменений модели данных. Если нужно поддерживать только одну страну, то данные в примере приложения могут благополучно пользоваться номерами карточек социального страхования, но обеспечение поддержки других стран может потребовать удаления или модификации свойства `SSN`. Когда суррогатный ключ не применяется, сделать это гораздо труднее.

Обеспечение уникальных значений для естественных ключей

Хотя для уникальной идентификации объектов `Employee` естественный ключ не используется, по-прежнему важно гарантировать отсутствие дублированных значений в свойстве `SSN`, хранящемся в БД. Это поможет предотвратить неправильный ввод пользователями записей и свести к минимуму проблемы, когда позже понадобится изменить модель данных. Простейший способ предотвращения дубликатов предусматривает создание индекса для свойства (листинг 19.18).

Листинг 19.18. Создание индекса в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) { }
    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .Property(e => e.Id).ForSqlServerUseSequenceHiLo();
      modelBuilder.Entity<Employee>()
        .HasIndex(e => e.SSN).HasName("SSNIndex").IsUnique();
    }
  }
}
```

Индексы создаются путем выбора класса с применением метода `Entity()` и вызова метода `HasIndex()` с целью выбора свойства, для которого будет создан индекс. Метод `HasName()` используется для указания имени индекса, а при работе с естественными ключами потребуется также вызвать метод `IsUnique()`, чтобы добавить в БД ограничение, запрещающее дублированные значения.

На заметку! Индексы могут создаваться только с помощью Fluent API. Поддержка со стороны атрибутов для средства индексов отсутствует.

В листинге 19.18 был настроен уникальный индекс для свойства `SSN`, который можно добавить к БД, создав и применив миграцию с использованием команд из листинга 19.19; команды должны выполняться в папке проекта `AdvancedApp`.

Листинг 19.19. Создание и применение миграции к БД

```
dotnet ef migrations add UniqueIndex
dotnet ef database update
```

Если вы просмотрите метод `Up()` в файле `<отметка времени>_UniqueIndex.cs`, добавленном в папку `Migrations`, то увидите, каким образом оператор Fluent API в листинге 19.18 изменил БД для обеспечения уникальности естественного ключа:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.AlterColumn<string>(name: "SSN",
    table: "Employees", type: "nvarchar(450)", nullable: true,
    oldClrType: typeof(string), oldNullable: true);

  migrationBuilder.CreateIndex(name: "SSNIndex",
    table: "Employees", column: "SSN", unique: true,
    filter: "[SSN] IS NOT NULL");
}
...
```

Первый оператор в методе `Up()` изменяет тип данных столбца `SSN`, чтобы он имел фиксированный размер. Второй оператор создает индекс, устанавливая аргумент `unique` в `true`, так что дублированные записи запрещены. Запустите приложение, перейдите по ссылке `http://localhost:5000` и попробуйте сохранить новый объект `Employee` со значением свойства `SSN`, которое уже есть у существующего объекта; вы получите сообщение об ошибке, представленное на рис. 19.5.

Создание альтернативного ключа

Когда необходимо создать отношения, используя значения естественного ключа, требуется другой подход. В подобных ситуациях нужен *альтернативный ключ*, который гарантирует уникальные значения и также конфигурирует БД, чтобы объект мог уникальным образом идентифицироваться по дополнительному ключу наравне с первичным ключом.

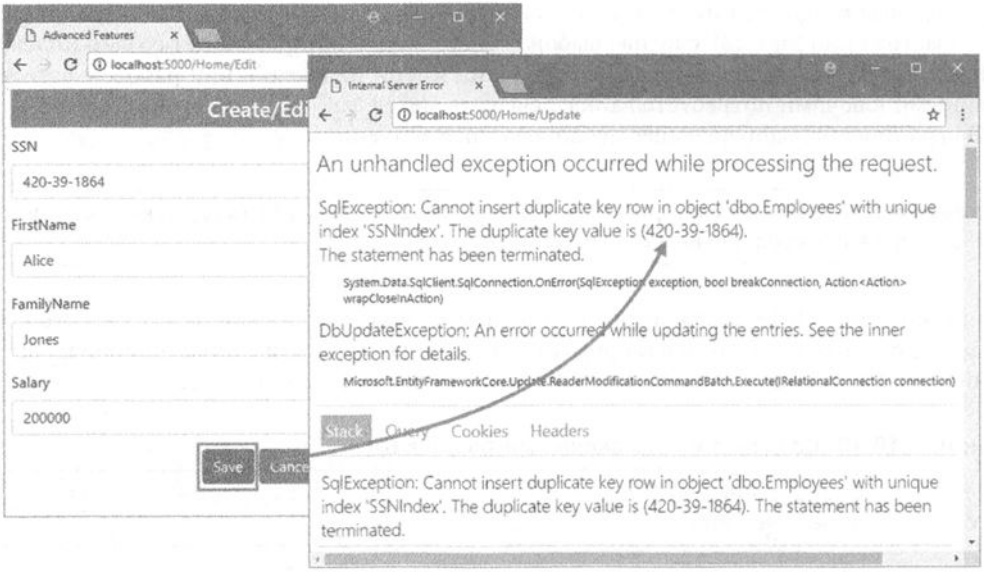

Рис. 19.5. Обеспечение уникальных значений для естественного ключа

Выяснение, когда альтернативные ключи полезны

В большинстве приложений можно безопасно создавать отношения с применением первичного ключа, что инфраструктура Entity Framework Core по умолчанию и будет делать. Если вам необходимо только избежать дублированных значений, тогда создайте уникальный индекс, а не альтернативный ключ, как было описано в предыдущем разделе. Возможность создания отношения на альтернативном ключе важна, только если вы ожидаете, что в будущем значение альтернативного ключа переместится на другой объект, и хотите гладко перенести существующие отношения, о чем в большинстве приложений заботиться не нужно.

Чтобы продемонстрировать использование альтернативного ключа, добавьте в папку `Models` файл по имени `SecondaryIdentity.cs` с определением класса, приведенным в листинге 19.20.

Листинг 19.20. Содержимое файла `SecondaryIdentity.cs` из папки `Models`

```
namespace AdvancedApp.Models {

  public class SecondaryIdentity {
    public long Id { get; set; }
    public string Name { get; set; }
    public bool InActiveUse { get; set; }

    public string PrimarySSN { get; set; }
    public Employee PrimaryIdentity { get; set; }
  }
}
```

Класс SecondaryIdentity будет представлять еще одно имя, под которым известен служащий. В нем определено свойство PrimaryIdentity, формирующее часть отношения с классом Employee, и свойство PrimarySSN, которое планируется применять в качестве внешнего ключа. Для укомплектования отношения добавьте в класс Employee обратное навигационное свойство (листинг 19.21).

Листинг 19.21. Укомплектование отношения в файле Employee.cs из папки Models

```
namespace AdvancedApp.Models {
  public class Employee {
    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }
    public decimal Salary { get; set; }
    public SecondaryIdentity OtherIdentity { get; set; }
  }
}
```

Свойство OtherIdentity возвращает объект SecondaryIdentity, который сообщает инфраструктуре Entity Framework Core о том, что это отношение "один к одному".

По умолчанию в отношении между классами SecondaryIdentity и Employee для столбца внешнего ключа инфраструктура Entity Framework Core будет использовать первичный ключ класса Employee. Добавьте в класс контекста операторы Fluent API, показанные в листинге 19.22, чтобы переопределить такое соглашение и взамен применять альтернативный ключ.

На заметку! Альтернативные ключи могут создаваться только с помощью Fluent API. Поддержка со стороны атрибутов для средства альтернативных ключей отсутствует.

Листинг 19.22. Использование альтернативного ключа в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {
  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) { }
    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .Property(e => e.Id).ForSqlServerUseSequenceHiLo();
      // modelBuilder.Entity<Employee>()
      //   .HasIndex(e => e.SSN).HasName("SSNIndex").IsUnique();
      modelBuilder.Entity<Employee>().HasAlternateKey(e => e.SSN);
```

```
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => e.SSN)
        .HasForeignKey<SecondaryIdentity>(s => s.PrimarySSN);
    }
  }
}
```

Первый новый оператор в листинге 19.22 создает альтернативный ключ с применением метода `HasAlternateKey()`, который дает тот же самый эффект, что и создание уникального индекса, но инфраструктура Entity Framework Core также разрешит создавать отношения, используя выбранное свойство. Метод `HasAlternateKey()` требуется для подготовки свойства как альтернативного ключа, только если вы не собираетесь настраивать отношение немедленно, но я включаю его вызов в любом случае, просто чтобы сделать свои намерения очевидными.

Второй новый оператор в листинге 19.22 устанавливает отношение между двумя классами. Методы `HasOne()` и `WithOne()` применяются для выбора навигационных свойств, а методы `HasPrincipalKey<T>()` и `HasForeignKey<T>()` — для выбора свойств альтернативного и внешнего ключей.

Результатом будет конфигурирование свойства `SSN` в качестве альтернативного ключа, который используется как внешний ключ в отношении с классом `SecondaryIdentity`. Выполните в папке проекта `AdvancedApp` команды из листинга 19.23 для создания и применения миграции к БД.

Листинг 19.23. Создание и применение миграции к БД

```
dotnet ef migrations add AlternateKey
dotnet ef database update
```

Если вы просмотрите метод `Up()` в файле `<отметка времени>_AlternateKey.cs`, добавленном в папку `Migrations`, то заметите ограничение внешнего ключа, примененное к столбцу `PrimarySSN` в таблице, которая была создана для хранения объектов `SecondaryIdentity`:

```
...
constraints: table => {
  table.PrimaryKey("PK_SecondaryIdentity", x => x.Id);
  table.ForeignKey(
    name: "FK_SecondaryIdentity_Employees_PrimarySSN",
    column: x => x.PrimarySSN,
    principalTable: "Employees",
    principalColumn: "SSN",
    onDelete: ReferentialAction.Restrict);
});
...
```

После определения альтернативный ключ можно использовать для создания отношений, как если бы он был первичным ключом. Чтобы внести изменения в часть ASP.NET Core MVC приложения, добавьте в представление `Edit.cshtml` элементы, выделенные в листинге 19.24 полужирным, которые позволят пользователю создавать или редактировать объект `SecondaryIdentity` наряду со связанным объектом.

Листинг 19.24. Добавление элементов в файле `Edit.cshtml` из папки `Views/Home`

```
@model Employee
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h4 class="bg-info p-2 text-center text-white">
  Create/Edit
</h4>
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <div class="form-group">
    <label class="form-control-label" asp-for="SSN"></label>
    <input class="form-control" asp-for="SSN" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FirstName"></label>
    <input class="form-control" asp-for="FirstName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FamilyName"></label>
    <input class="form-control" asp-for="FamilyName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="Salary"></label>
    <input class="form-control" asp-for="Salary" />
  </div>
  <input type="hidden" asp-for="OtherIdentity.Id" />
  <div class="form-group">
    <label class="form-control-label">Other Identity Name:</label>
    <input class="form-control" asp-for="OtherIdentity.Name" />
  </div>
  <div class="form-check">
    <label class="form-check-label">
    <input class="form-check-input" type="checkbox"
          asp-for="OtherIdentity.InActiveUse" />
      In Active Use
    </label>
  </div>
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Save</button>
    <a class="btn btn-secondary" asp-action="Index">Cancel</a>
  </div>
</form>
```

Модифицируйте также метод `Edit()` в контроллере `Home`, чтобы запрос объекта `Employee` следовал по навигационному свойству для включения связанных данных и передавал их представлению (листинг 19.25).

Листинг 19.25. Включение связанных данных в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {
  public class HomeController : Controller {
    private AdvancedContext context;
    public HomeController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Employees);
    }
    public IActionResult Edit(long id) {
      return View(id == default(long)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.Id == id));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (employee.Id == default(long)) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Запрос в методе `Edit()` применяет метод `Include()` для следования по навигационному свойству и метод `First()` для нахождения объекта со значением `Id`, указанным пользователем. Чтобы проверить, работоспособен ли альтернативный ключ, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и создайте или отредактируйте объект `Employee`. Благодаря добавлению нового отношения (и применяемого им альтернативного ключа) вы можете предоставить детали второй идентичности (рис. 19.6).

Совет. Не вносите никаких изменений в свойства `SSN`, `FirstName` или `FamilyName` объектов, которые не были сохранены в БД. Свойства, формирующие ключ, не могут быть изменены, если только БД специально не сконфигурирована, чтобы разрешать это.

Использование естественных ключей как первичных ключей

Если вы не хотите применять суррогатный ключ (обычно из-за того, что уверены в отсутствии изменений модели данных в будущем), тогда можете выбрать свойства для использования в качестве первичного ключа и возложить на себя ответственность за

генерирование уникальных значений, хотя такое решение не должно приниматься необдуманно, поскольку вам придется отвечать за обеспечение уникальности каждого значения ключа. Естественные ключи могут быть неаккуратными и не всегда можно полагаться, что они настолько уникальны, насколько казалось на стадии проектирования. Для демонстрации применения естественного ключа как первичного ключа измените конфигурацию модели данных, используя операторы Fluent API, которые сообщают инфраструктуре Entity Framework Core о том, что существующее свойство первичного ключа должно игнорироваться, а взамен применяться свойство `SSN` (листинг 19.26).

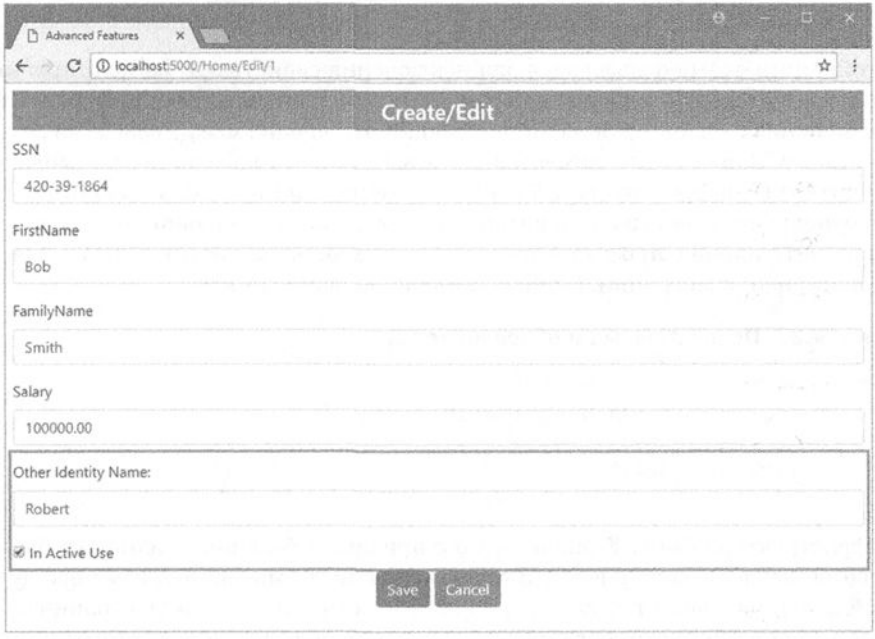

Рис. 19.6. Использование альтернативного ключа как внешнего ключа в отношении

На заметку! За счет использования атрибута `Key` в качестве первичного ключа можно выбирать любое свойство. Остаток примера будет тем же, в том числе изменения, необходимые для работы с ключами, за генерацию которых несет ответственность пользователь/приложение.

Листинг 19.26. Применение естественного ключа в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }
```

```
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>().HasKey(e => e.SSN);
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => e.SSN)
        .HasForeignKey<SecondaryIdentity>(s => s.PrimarySSN);
    }
  }
}
```

Метод `Ignore()` использовался для исключения свойства `Id` из модели данных, а метод `HasKey()` — для выбора свойства `SSN` как первичного ключа. Вносить какие-либо изменения в оператор, конфигурирующий отношение между классами `Employee` и `SecondaryIdentity`, не требуется, хотя вызов `HasPrincipalKey()` можно было бы удалить, т.к. инфраструктура Entity Framework Core по умолчанию будет применять в отношении свойство `SSN`, которое теперь является первичным ключом.

Выполните в папке проекта `AdvancedApp` команды из листинга 19.27 для создания и применения миграции, которая изменит первичный ключ.

Листинг 19.27. Переустановка и обновление БД

```
dotnet ef migrations remove --force
dotnet ef migrations add NaturalPrimaryKey
dotnet ef database drop --force
dotnet ef database update
```

Инфраструктура Entity Framework Core прилагает большие усилия, чтобы внести изменения, вовлекающие ключи, и попытка добавить миграцию, изменяющую первичный ключ, не сработает, т.к. миграция будет стараться удалить ограничение, настроенное для альтернативного ключа, которое используется созданным на нем отношением. Для обхода проблемы миграция, настраивающая альтернативный ключ, удаляется, затем создается миграция, которая выбирает свойство `SSN` в качестве первичного ключа, и, наконец, БД воссоздается.

Чтобы учесть изменения в части ASP.NET Core MVC приложения, обновите контроллер для применения свойства `SSN` как первичного ключа (листинг 19.28).

Листинг 19.28. Использование нового первичного ключа в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;
```

```
    public IActionResult Index() {
      return View(context.Employees);
    }
    public IActionResult Edit(string SSN) {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN) == 0) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Самое важное изменение касается метода `Update()`. Когда за генерацию значений ключа отвечал сервер баз данных, то определять, являлся запрос операцией обновления или создания, можно было путем проверки на предмет стандартного значения для типа ключа. Теперь за предоставление значения первичного ключа несет ответственность пользователь, поэтому такой прием не годится. Взамен у БД выясняется, существует ли объект со значением ключа, помещенным в запрос, что делается с помощью LINQ-метода `Count()`.

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create и сохраните новый объект `Employee` в БД. Просмотрев журнальные сообщения, которые сгенерированы приложением, вы увидите, что запрос в методе `Update()` превратился в следующую операцию:

```
...
SELECT COUNT(*)
FROM [Employees] AS [e]
WHERE [e].[SSN] = @__employee_SSN_0
...
```

Такая проверка позволяет выяснить, присутствует ли в БД ключ, указанный пользователем, без необходимости в загрузке данных и столкновении с кешем данных Entity Framework Core при выполнении обновления с участием объекта, который создается связывателем моделей MVC.

Создание составных ключей

Составной ключ уникальным образом идентифицирует объект за счет объединения значений из двух и более столбцов в таблице БД или свойств из сущностного класса. Создавать составные ключи при использовании для генерации значений ключей стратегии Identity или Hi-Lo не понадобится, потому что получаемые значения всегда будут уникальными, но составной ключ может быть полезен, когда применяется естественный ключ, который уникален только в комбинации с другим значением.

Номера карточек социального страхования часто трактуются в США как уникальные, но ряд исследований показали, что около 40 миллионов номеров используются более чем одним лицом из-за сочетания путаницы, ошибок и мошенничества. В настоящий момент пример приложения сгенерирует исключение при попытке создать объект `Employee`, в котором применяется значение `SSN`, уже хранящееся в БД, но в этом разделе правила об уникальности `SSN` будут ослаблены за счет идентификации объектов посредством комбинации свойств (листинг 19.29).

На заметку! Составные ключи могут создаваться только с помощью Fluent API. Поддержка со стороны атрибутов для средства составных ключей отсутствует.

Листинг 19.29. Создание составного ключа в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
    }
  }
}
```

Составные ключи создаются путем создания объекта, выбирающего свойства, которые должны использоваться в ключе (в примере это свойства `SSN`, `FirstName` и `FamilyName`). Для каждого из указанных свойств в БД могут присутствовать дублированные значения наряду с тем, что каждый объект имеет уникальную комбинацию значений. Свойства, образующие ключ, также должны применяться для создания отношений, отраженных в лямбда-выражениях, которые используются в методе `HasPrincipalKey()`, конфигурирующем одну сторону отношения с классом `SecondaryIdentity`:

```
...
.HasPrincipalKey<Employee>(e =>
    new { e.SSN, e.FirstName, e.FamilyName })
...
```

Для метода HasForeignKey() были указаны дополнительные свойства внешнего ключа, чтобы отслеживать связанный объект Employee с применением составного ключа; эти свойства определены в классе SecondaryIdentity (листинг 19.30).

Листинг 19.30. Добавление свойств внешнего ключа в файле SecondaryIdentity.cs из папки Models

```
namespace AdvancedApp.Models {

  public class SecondaryIdentity {
    public long Id { get; set; }
    public string Name { get; set; }
    public bool InActiveUse { get; set; }

    public string PrimarySSN { get; set; }
    public string PrimaryFamilyName { get; set; }
    public string PrimaryFirstName { get; set; }
    public Employee PrimaryIdentity { get; set; }
  }
}
```

Выполните в папке проекта AdvancedApp команды из листинга 19.31 для создания новой миграции и ее применения к БД. Инфраструктура Entity Framework Core приложит усилия, чтобы внести изменения, требующиеся для первичного ключа, а потому команды удалят миграцию из предыдущего раздела, добавят новую миграцию и воссоздадут БД.

Листинг 19.31. Создание и применение миграции к БД

```
dotnet ef migrations remove --force
dotnet ef migrations add CompositeKey
dotnet ef database drop --force
dotnet ef database update
```

Исследуя метод Up() в файле <отметка времени>_CompositeKey.cs внутри папки Migrations, вы обнаружите оператор, который будет конфигурировать первичный ключ для таблицы Employees с использованием комбинации свойств, выбранной в листинге 19.29:

```
...
migrationBuilder.AddPrimaryKey(
  name: "PK_Employees",
  table: "Employees",
  columns: new[] { "SSN", "FirstName", "FamilyName" });
...
```

Когда создается составной ключ, то требуется внести изменение в оставшиеся части приложения. Обновите контроллер, чтобы при запрашивании БД он работал со всеми свойствами ключа (листинг 19.32).

Листинг 19.32. Применение составного ключа в файле HomeController.cs из папки Controllers

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
```

```
using System.Linq;
namespace AdvancedApp.Controllers {
  public class HomeController : Controller {
    private AdvancedContext context;
    public HomeController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Employees);
    }
    public IActionResult Edit(string SSN, string firstName,
                              string familyName) {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Без таких изменений контроллер не будет запрашивать БД, используя полный первичный ключ, что приведет в несколько странным результатам — либо выбор неправильного объекта для редактирования, либо отказ вставки нового объекта, если он имеет значение SSN, которое уже существует в БД.

Совет. Некоторые методы, такие как Find(), принимают последовательность значений ключа, применяемых для запрашивания БД. В случае использования подобных методов значения должны предоставляться в том же порядке, который применялся при определении составного ключа в классе контекста. В примере приложения это означает использование значений свойств SSN, FirstName и FamilyName в указанном порядке, поскольку именно так определялся составной ключ.

Финальное изменение связано с добавлением в представление Index.cshtml ряда дополнительных атрибутов, которые выбирают объект для редактирования, чтобы метод Edit() контроллера принимал все значения первичного ключа (листинг 19.33).

Листинг 19.33. Применение составного ключа в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
   ViewData["Title"] = "Advanced Features";
   Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>Key</th>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="7" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.Id</td>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td class="text-right">
          <a asp-action="Edit" asp-route-ssn="@e.SSN"
             asp-route-firstname="@e.FirstName"
             asp-route-familyname="@e.FamilyName"
             class="btn btn-sm btn-primary">Edit</a>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Атрибуты `asp-route-` предоставляют значения составного ключа, требуемые для идентификации объекта, когда пользователь редактирует `Employee`. Чтобы удостовериться в работоспособности составного ключа, запустите приложение, перейдите по ссылке `http://localhost:5000` и создайте новые объекты `Employee`, используя значения данных из табл. 19.4.

Таблица 19.4. Данные для проверки составного ключа

SSN (**Номер карточки социального страхования**)	First Name (**Имя**)	Family Name (**Фамилия**)	Salary (**Оклад**)	Other Name (**Другое имя**)	In Active Use (**Активно используется**)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
420-39-1864	Alice	Jones	200000	Allie	Флажок отмечен
420-39-1864	Bob	Smith	150000	Bobby	Флажок не отмечен

Все три строки в таблице содержат одно и то же значение для свойства `SSN`, но вы получите исключение только при попытке создания третьего объекта, который содержит ту же самую комбинацию значений для трех свойств, применяемых в первичном ключе (рис. 19.7).

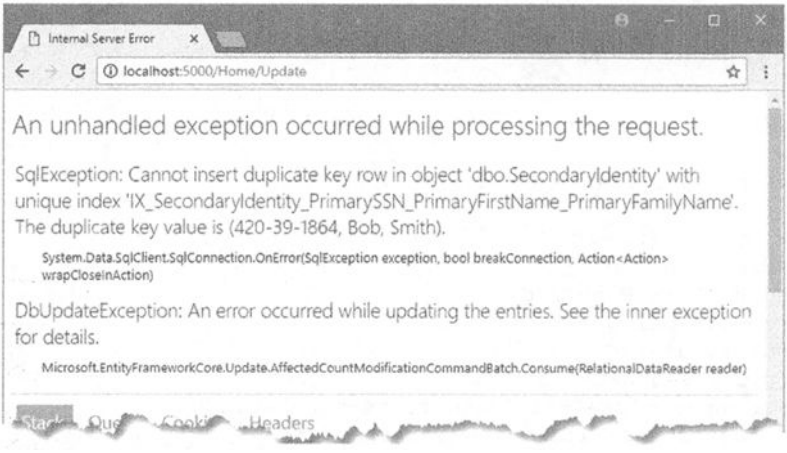

Рис. 19.7. Попытка создания дублированного составного ключа

Сообщение об ошибке включает детали дублированного ключа, которые показывают, что он является комбинацией свойств первичного ключа, используемого для идентификации объектов:

```
...
The duplicate key value is (420-39-1864, Bob, Smith).
Значение дублированного ключа: (420-39-1864, Bob, Smith).
...
```

Когда комбинация значений формирует ключ, отдельные свойства могут дублироваться, как демонстрировалось в примере.

Резюме

В главе обсуждались расширенные возможности инфраструктуры Entity Framework Core, предназначенные для работы с ключами. Объяснялись стратегии генерации ключей Identity и Hi-Lo и работа с естественными ключами разнообразными способами: обеспечение уникальных значений, применение их для создания отношений и использование в качестве первичных ключей. В следующей главе будут описаны расширенные возможности для запросов.

ГЛАВА 20

Запросы

Поддержка запросов с помощью LINQ в Entity Framework Core превращает взаимодействие с данными в естественный процесс для разработчиков. В этой главе будут описаны расширенные функции, предлагаемые инфраструктурой Entity Framework Core для управления запросами, которые могут быть полезны, если требуемое поведение не удается получить с использованием приемов, рассмотренных в предшествующих главах. В табл. 20.1 приведены сведения, позволяющие поместить расширенные функции запросов в контекст.

Таблица 20.1. Помещение расширенных функций запросов в контекст

Вопрос	Ответ
Что это такое?	Расширенные функции запросов позволяют переопределять стандартное поведение Entity Framework Core
Чем они полезны?	Расширенные функции запросов могут быть полезны при работе с существующими БД или когда имеются специфические требования к производительности
Как они используются?	Расширенные функции запросов применяются как часть запросов LINQ
Существуют ли какие-то скрытые ловушки или ограничения?	Расширенные функции запросов могут неожиданно изменять поведение приложений или результаты запросов и должны использоваться с осторожностью
Существуют ли альтернативы?	Это специализированные средства, которые не требуются в большинстве проектов

В табл. 20.2 приведена сводка по главе.

Таблица 20.2. Сводка по главе

Задача	Решение	Листинг
Запрашивание данных только для чтения	Отключите средство отслеживания изменений	20.1–20.7
Фильтрация данных, вырабатываемых всеми запросами	Примените фильтр запросов	20.8–20.12
Переопределение фильтра запросов	Используйте метод `IgnoreQueryFilters()`	20.13, 20.14
Запрашивание данных с применением поискового выражения	Используйте функцию `Like()`	20.15
Выполнение параллельных запросов	Применяйте асинхронные методы запросов	20.16, 20.17
Ускорение повторного использования запроса	Явно скомпилируйте запрос	20.18
Распознавание клиентской оценки запросов	Включите отчеты об исключениях, которые будут генерироваться, когда запросы содержат клиентскую оценку	20.19, 20.21

Подготовительные шаги

В главе продолжается работа с проектом `AdvancedApp`, который был создан в главе 19. В качестве подготовки модифицируйте представление, применяемое для создания и редактирования объектов `Employee`, чтобы значения составного первичного ключа не могли изменяться (листинг 20.1).

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 20.1. Запрет изменения ключа в файле `Edit.cshtml` из папки `Views/Home`

```
@model Employee
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}

<h4 class="bg-info p-2 text-center text-white">
  Create/Edit
</h4>
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <div class="form-group">
    <label class="form-control-label" asp-for="SSN"></label>
    <input class="form-control" asp-for="SSN" readonly="@Model.SSN" />
  </div>
  <div class="form-group">
```

```
    <label class="form-control-label" asp-for="FirstName"></label>
    <input class="form-control" asp-for="FirstName"
        readonly="@Model.FirstName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FamilyName"></label>
    <input class="form-control" asp-for="FamilyName"
        readonly="@Model.FamilyName"/>
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="Salary"></label>
    <input class="form-control" asp-for="Salary" />
  </div>
  <input type="hidden" asp-for="OtherIdentity.Id" />
  <div class="form-group">
    <label class="form-control-label">Other Identity Name:</label>
    <input class="form-control" asp-for="OtherIdentity.Name" />
  </div>
  <div class="form-check">
    <label class="form-check-label">
    <input class="form-check-input" type="checkbox"
        asp-for="OtherIdentity.InActiveUse" />
      In Active Use
    </label>
  </div>
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Save</button>
    <a class="btn btn-secondary" asp-action="Index">Cancel</a>
  </div>
</form>
```

Выполните в папке проекта AdvancedApp команды из листинга 20.2 для удаления и воссоздания БД.

Листинг 20.2. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение, используя dotnet run, перейдите по ссылке http://localhost:5000, щелкните на кнопке Create (Создать) и сохраните три объекта Employee с применением значений, приведенных в табл. 20.3.

Результат создания трех объектов Employee показан на рис. 20.1.

Таблица 20.3. Значения данных для создания объектов в примере

SSN (Номер карточки социального страхования)	First Name (Имя)	Family Name (Фамилия)	Salary (Оклад)	Other Name (Другое имя)	In Active Use (Активно используется)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
657-03-5898	Alice	Jones	200000	Allie	Флажок отмечен
300-30-0522	Peter	Davies	180000	Pete	Флажок отмечен

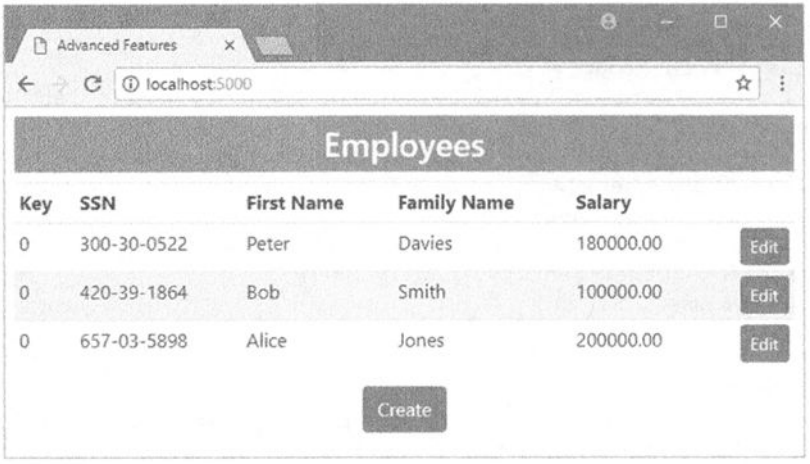

Рис. 20.1. Выполнение примера приложения

Управление отслеживанием изменений для результатов, производимых запросами

Средство отслеживания изменений представляет собой одну из функций, которые делают инфраструктуру Entity Framework Core удобной в использовании. При поступлении запроса к БД инфраструктура Entity Framework Core начинает отслеживать объекты, которые она создает для представления данных. Когда вызывается метод `SaveChanges()`, инфраструктура Entity Framework Core идентифицирует свойства, значения которых изменились, и надлежащим образом обновляет БД.

Каким бы ни было полезным это средство, оно требуется далеко не для всех запросов, выполняемых в приложениях ASP.NET Core MVC. Многие HTTP-запросы, получаемые приложением MVC, нацелены на методы действий, которые лишь читают данные из БД и ничего не изменяют. В случае чтения данных никакой выгоды от работы, которую инфраструктура должна проделать для настройки отслеживания изменений в создаваемых объектах, извлечь не удастся, поскольку изменения никогда не произойдут.

Вы можете управлять тем, выполняется ли отслеживание изменений для запроса, с применением методов, перечисленных в табл. 20.4, которые вызываются на объектах реализации `IQueryable<T>`.

Таблица 20.4. Методы, используемые для настройки отслеживания изменений

Имя	Описание
`AsNoTracking()`	Отключает отслеживание изменений для результатов запроса, к которому применяется
`AsTracking()`	Включает отслеживание изменений для результатов запроса, к которому применяется

Методы, описанные в табл. 20.4, применяются для управления отслеживанием изменений в индивидуальных запросах. По умолчанию отслеживание изменений включено, поэтому отключите его для запросов, выполняющих только чтение, в контроллере `Home` (листинг 20.3).

Листинг 20.3. Отключение отслеживания изменений в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(context.Employees.AsNoTracking());
    }

    public IActionResult Edit(string SSN, string firstName,
                              string familyName) {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .AsNoTracking()
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }

    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

В запросы внутри методов действий `Index()` и `Edit()` добавлены вызовы метода `AsNotTracking()`. Методы `AsTracking()` и `AsNotTracking()` применяются к объектам реализации `IQueryable<T>`, т.е. они должны включаться в цепочку методов, создающих запрос, перед методами вроде `First()`, которые сужают результат до одиночного объекта.

Отключение отслеживания изменений не дает какого-то заметного эффекта, но на самом деле инфраструктура Entity Framework Core больше не настраивает отслеживание для объектов, которые используются в действиях, только читающих БД.

Исключение индивидуальных объектов из отслеживания изменений

Распространенная проблема с отслеживанием изменений в приложениях ASP.NET Core MVC возникает при попытке выполнить обновление с применением объекта, созданного связывателем моделей MVC, который имеет такой же первичный ключ, как у объекта, загруженного запросом Entity Framework Core, где используется отслеживание изменений. Для демонстрации проблемы модифицируйте метод `Update()` в контроллере `Home` согласно листингу 20.4.

Листинг 20.4. Смешанные объекты в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(Employee employee) {
  if (context.Employees.Find(employee.SSN, employee.FirstName,
      employee.FamilyName) == null) {
    context.Add(employee);
  } else {
    context.Update(employee);
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Запрос в листинге 20.4 изменен так, что он применяет метод `Find()` для выяснения, использовался ли уже ключ. Это искусственная проблема, поскольку первоначальный код показал, что передача лямбда-выражения LINQ-методу `Count()` работала без каких-либо проблем, но она настолько распространена, что ее стоит продемонстрировать, несмотря на представление одного приема, который позволяет ее избежать.

Запустите приложение с помощью `dotnet run` и перейдите по ссылке `http://localhost:5000`. Щелкните на кнопке Edit (Редактировать) для любого объекта `Employee` и затем на кнопке Save (Сохранить); появится сообщение об ошибке (рис. 20.2).

В сообщении об ошибке указано, что отслеживаться может только один объект с конкретным ключом. Проблема возникла из-за того, что инфраструктура Entity Framework Core поместила объект `Employee`, созданный в качестве результата выполнения метода `Find()`, под контроль отслеживания изменений, и он имеет такой же первичный ключ, как у объекта `Employee`, который был создан связывателем моделей MVC и передан методу `Update()`.

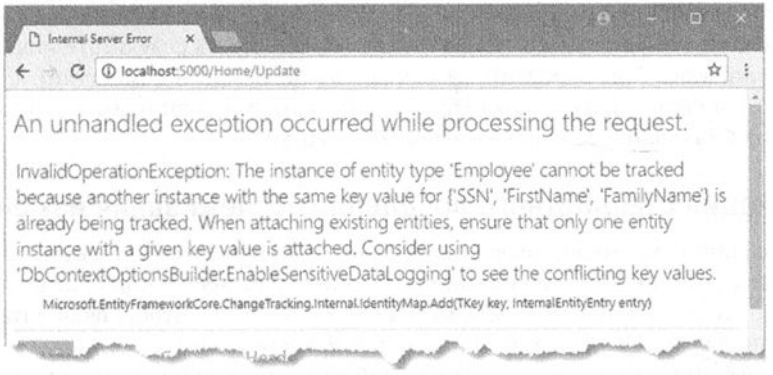

Рис. 20.2. Исключение, вызванное средством отслеживания изменений

Инфраструктура Entity Framework Core не может отслеживать изменения для двух объектов с одинаковыми первичными ключами, потому что не в состоянии урегулировать любые конфликтующие изменения, которые в них вносятся, а потому она генерирует исключение.

Избежать такой проблемы легче всего за счет применения запросов, выпускающих простые результаты, которые не подпадают под отслеживание изменений, таких как использовались в первоначальном коде, опирающемся на LINQ-метод `Count()`. Инфраструктура Entity Framework Core выполняет отслеживание изменений только для сущностных объектов, поэтому любой запрос, дающий несущностный результат наподобие значения `int`, не подлежит отслеживанию изменений. Можно также применить описанный в предыдущем разделе метод `AsNoTracking()`, который исключит все объекты, созданные запросом, из отслеживания изменений.

Если ни один из указанных приемов не подходит, тогда можно явно исключить объект из отслеживания изменений. Это не отменит работу Entity Framework Core, выполняемую для отслеживания объекта, но предотвратит генерацию исключения.

Модифицируйте метод `Update()` так, чтобы исключить объект `Employee`, создаваемый инфраструктурой Entity Framework Core, из отслеживания изменений и устранить возможность его конфликта с объектом `Employee`, который создается связывателем моделей MVC (листинг 20.5).

Листинг 20.5. Исключение объекта из отслеживания изменений в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(Employee employee) {
  Employee existing = context.Employees.Find(employee.SSN,
    employee.FirstName, employee.FamilyName);
  if (existing == null) {
    context.Add(employee);
  } else {
    context.Entry(existing).State = EntityState.Detached;
    context.Update(employee);
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Отслеживаемый объект передается методу `Entry()` объекта контекста, а свойству `State` присваивается значение `EntityState.Detached`. В результате инфраструктура Entity Framework Core исключит объект из отслеживания изменений, т.е. он больше не будет конфликтовать с объектом с тем же самым первичным ключом, который создается из HTTP-запроса связывателем моделей MVC.

Модификация стандартного поведения отслеживания изменений

Если большинство имеющихся запросов не модифицируют объекты, тогда может быть проще отключить отслеживание изменений для всех запросов, выполняемых объектом контекста, и использовать метод `AsTracking()`, чтобы включать отслеживание только для тех запросов, которые в нем нуждаются.

В листинге 20.6 отслеживание изменений отключается для всех запросов в классе `AdvancedContext`. В примере приложения присутствует только один контекст, но отключение отслеживания изменений не повлияет на другие контексты, каждый из которых пришлось бы конфигурировать аналогичным образом.

Листинг 20.6. Отключение отслеживания изменений в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

    public AdvancedContext(DbContextOptions<AdvancedContext> options)
        : base(options) {
      ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
    }

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
    }
  }
}
```

Свойство `ChangeTracker`, определенное в классе `DbContext`, возвращает объект `ChangeTracker`, свойство `QueryTrackingBehavior` которого конфигурируется с применением перечисления с таким же именем. Значения перечисления `QueryTrackingBehavior` описаны в табл. 20.5.

Таблица 20.5. Значения `QueryTrackingBehavior`

Имя	Описание
`NoTracking`	Отключает отслеживание изменений для запросов, выполняемых объектом контекста
`TrackAll`	Включает отслеживание изменений для запросов, выполняемых объектом контекста

Если по умолчанию отслеживание изменений отключено, то в любом запросе, полагающемся на отслеживание с целью обнаружения изменений, должен использоваться метод `AsTracking()`. Модифицируйте запрос в методе `Update()` контроллера `Home`, чтобы значение свойства `Salary` применялось к объекту `Employee`, прочитанному из БД, что будет работать, только если инфраструктура Entity Framework Core разрешает использовать отслеживание для обнаружения измененного значения (листинг 20.7).

Листинг 20.7. Включение отслеживания изменений для запроса в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(Employee employee) {
  Employee existing = context.Employees
    .AsTracking()
    .First(e => e.SSN == employee.SSN && e.FirstName == employee.FirstName
      && e.FamilyName == employee.FamilyName);
  if (existing == null) {
    context.Add(employee);
  } else {
    existing.Salary = employee.Salary;
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Без вызова метода `AsTracking()` инфраструктура Entity Framework Core не смогла бы обнаружить изменение и не обновила бы БД.

Использование фильтра запросов

Фильтр запросов применяется ко всем запросам специфического сущностного класса, выполняемым в приложении. Одним полезным употреблением фильтра запросов является реализация функции "мягкого удаления", которая помечает удаляемые объекты, не удаляя их из БД, что позволяет восстановить данные, если они были удалены по неосторожности.

На заметку! Расширенные возможности для действительного удаления данных рассматриваются в главе 22.

В качестве подготовки добавьте в класс `Employee` свойство, которое будет указывать на то, что хранящийся в БД объект был мягко удален пользователем (листинг 20.8).

Листинг 20.8. Добавление свойства в файле `Employee.cs` из папки `Models`

```
namespace AdvancedApp.Models {
  public class Employee {
    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }
    public decimal Salary { get; set; }
    public SecondaryIdentity OtherIdentity { get; set; }
    public bool SoftDeleted { get; set; } = false;
  }
}
```

Далее добавьте фильтр запросов, который исключает мягко удаленные объекты `Employee` из результатов запросов, как показано в листинге 20.9. Кроме того, закомментируйте строку кода, отключающую отслеживание изменений, чтобы максимально упростить пример.

Листинг 20.9. Определение фильтра запросов в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
namespace AdvancedApp.Models {
  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
        : base(options) {
  // ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
    }
    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);
      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
    }
  }
}
```

Фильтры запросов создаются путем выбора класса с помощью метода `Entity()` и затем вызова метода `HasQueryFilter()`. Фильтр применяется ко всем запросам выбранных классов, и в результаты запросов будут включаться только те объекты, для которых лямбда-выражение возвращает `true`. В листинге 20.9 был определен фильтр запросов, выбирающий объекты `Employee`, свойство `SoftDeleted` которых имеет значение `false`.

Для реализации функции мягкого удаления обновите контроллер Home, как демонстрируется в листинге 20.10. Добавьте метод действия Delete(), который устанавливает свойство SoftDeleted объекта Employee в true, что заставит фильтр запросов исключать мягко удаленные объекты.

Листинг 20.10. Поддержка мягкого удаления в файле HomeController.cs из папки Controllers

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(context.Employees.AsNoTracking());
    }
   public IActionResult Edit(string SSN, string firstName, string familyName) {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
            .AsNoTracking()
            .First(e => e.SSN == SSN
              && e.FirstName == firstName
              && e.FamilyName == familyName));
    }

    [HttpPost]
    public IActionResult Update(Employee employee) {
      Employee existing = context.Employees
        .AsTracking()
        .First(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName);
      if (existing == null) {
        context.Add(employee);
      } else {
        existing.Salary = employee.Salary;
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Attach(employee);
      employee.SoftDeleted = true;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы дать пользователю возможность работать с функцией мягкого удаления, добавьте в представление `Index.cshtml` элементы, выделенные полужирным в листинге 20.11; они будут отправлять HTTP-запрос `POST`, содержащий значения первичного ключа объекта `Employee`, методу действия `Delete()`.

Листинг 20.11. Добавление элементов в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>Key</th>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="7" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.Id</td>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <button type="submit" asp-action="Delete" formmethod="post"
                  class="btn btn-sm btn-danger">Delete</button>
            <button type="submit" asp-action="Edit" formmethod="get"
                  class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Элемент `form` и его содержимое используются для функций удаления и редактирования. Каждый кнопочный элемент, заменяющий ранее использованный якорный элемент, сконфигурирован с действием и HTTP-методом, которые должны применяться, когда пользователь совершает щелчок на элементе.

Выполните в папке проекта `AdvancedApp` команды из листинга 20.12 для создания новой миграции и ее применения к БД.

Листинг 20.12. Создание и применение миграции к БД

```
dotnet ef migrations add SoftDelete
dotnet ef database update
```

Чтобы увидеть эффект мягкого удаления, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://locahost:5000` и удалите какой-нибудь объект `Employee`. После удаления объект исчезает из таблицы объектов `Employee`, как видно на рис. 20.3.

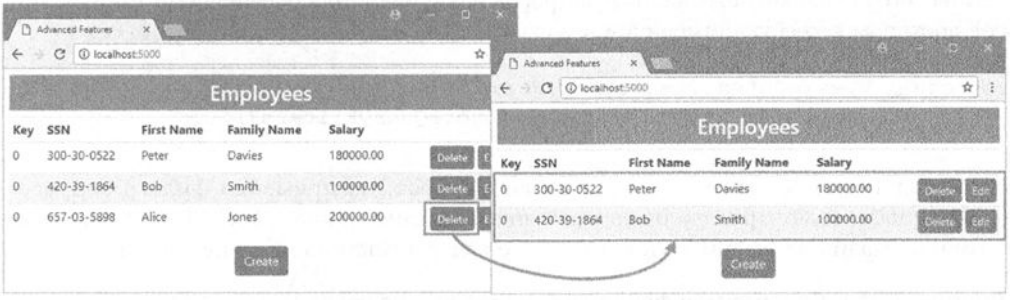

Рис. 20.3. Мягкое удаление данных

Переопределение фильтра запросов

Возможность мягкого удаления объектов полезна только при наличии средств для отмены удаления объектов. Это означает необходимость переопределения фильтра, чтобы можно было запрашивать у БД мягко удаленные объекты и отображать их пользователю. Добавьте в папку `Controllers` файл класса по имени `DeleteController.cs` с определением контроллера, представленным в листинге 20.13.

Листинг 20.13. Содержимое файла `DeleteController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class DeleteController : Controller {
    private AdvancedContext context;

    public DeleteController(AdvancedContext ctx) => context = ctx;
```

```
    public IActionResult Index() {
      return View(context.Employees.Where(e => e.SoftDeleted)
        .Include(e => e.OtherIdentity).IgnoreQueryFilters());
    }
    [HttpPost]
    public IActionResult Restore(Employee employee) {
      context.Employees.IgnoreQueryFilters()
        .First(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName).SoftDeleted = false;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Методы действий `Index()` и `Restore()` нуждаются в запрашивании мягко удаленных объектов, которые фильтр запросов исключает. Для обеспечения таких запросов доступом к требующимся данным вызывается метод `IgnoreQueryFilters()`:

```
...
return View(context.Employees.Where(e => e.SoftDeleted)
  .Include(e => e.OtherIdentity).IgnoreQueryFilters());
...
```

Метод `IgnoreQueryFilters()` делает запрос, не применяя фильтр запросов. Чтобы снабдить контроллер представлением, создайте папку `Views/Delete` и поместите в нее файл по имени `Index.cshtml` с содержимым из листинга 20.14.

Листинг 20.14. Содержимое файла `Index.cshtml` из папки `Views/Delete`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Deleted Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="4" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
```

```
        <td class="text-right">
          <form method="post">
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="FirstName" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <button asp-action="Restore"
                    class="btn btn-sm btn-success">Restore</button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
```

Запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000/delete`; вы увидите список мягко удаленных объектов. Щелкните на кнопке Restore (Восстановить) для установки свойства `SoftDeleted` объекта в `false`, что восстановит его в основной таблице данных, отображаемой контроллером `Home` (рис. 20.4).

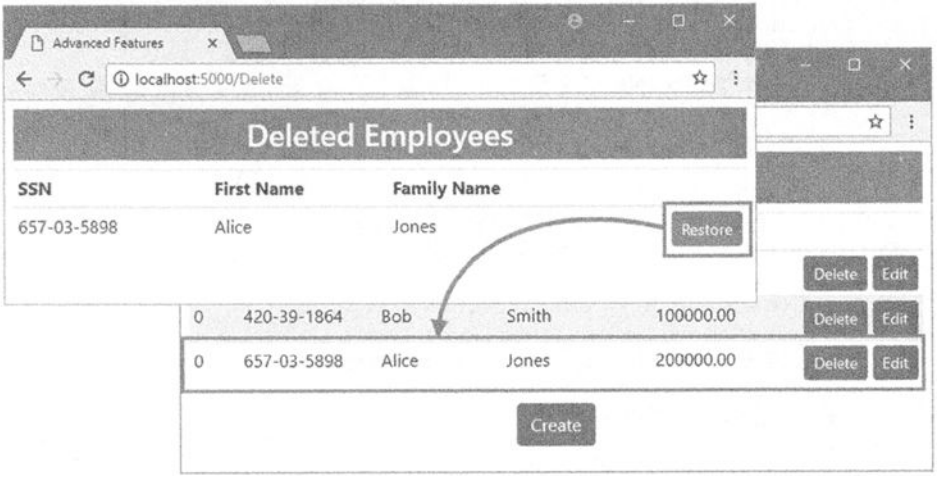

Рис. 20.4. Восстановление мягко удаленного объекта

Запрашивание с использованием поискового шаблона

Инфраструктура Entity Framework Core поддерживает SQL-выражение `LIKE`, что позволяет выполнять запросы с применением поисковых шаблонов. Модифицируйте метод действия `Index()` контроллера `Home`, чтобы он принимал параметр поискового термина, который используется для создания запроса с ключевым словом `LIKE` (листинг 20.15).

Листинг 20.15. Применение поискового шаблона в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index(string searchTerm) {
      IQueryable<Employee> data = context.Employees;
      if (!string.IsNullOrEmpty(searchTerm)) {
        data = data.Where(e => EF.Functions.Like(e.FirstName, searchTerm));
      }
      return View(data);
    }
    public IActionResult Edit(string SSN, string firstName, string familyName)
    {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .AsNoTracking()
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      Employee existing = context.Employees
        .AsTracking()
        .First(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName);
      if (existing == null) {
        context.Add(employee);
      } else {
        existing.Salary = employee.Salary;
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Attach(employee);
      employee.SoftDeleted = true;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Из-за отсутствия прямой поддержки LIKE в LINQ результирующий синтаксис довольно неуклюж. Метод EF.Functions.Like() используется для доступа к функциональности LIKE внутри конструкции Where() и принимает в качестве параметров свойство, которое будет сопоставляться, и поисковый термин. В листинге 20.15 метод Like() применялся для поиска объектов Employee, значение FirstName которых совпадает с поисковым термином, полученным в параметре метода действия. Поисковые термины могут выражаться с использованием четырех групповых символов, описанных в табл. 20.6.

Таблица 20.6. Групповые символы SQL-выражения LIKE

Групповой символ	Описание
%	Соответствует строке с нулем и более символов
_	Соответствует любому одиночному символу
[символы]	Соответствует любому одиночному символу, находящемуся в наборе
[^символы]	Соответствует любому одиночному символу, не находящемуся в наборе

Чтобы посмотреть, как работает поиск, запустите приложение с применением dotnet run, удостоверьтесь, что все объекты Employee были восстановлены после мягкого удаления, и перейдите по следующей ссылке:

```
http://localhost:5000?searchTerm=%[ae]%
```

Поисковый термин, указанный в строке запроса URL, будет соответствовать любому имени, которое содержит букву A или E. Если вы исследуете журнальные сообщения, сгенерированные приложением, то заметите запрос, который инфраструктура Entity Framework Core отправила серверу баз данных:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[Salary],
[e].[SoftDeleted]
FROM [Employees] AS [e]
WHERE ([e].[SoftDeleted] = 0) AND [e].[FirstName] LIKE @__searchTerm_1
...
```

Важная часть запроса, касающаяся ключевого слова LIKE, выделена полужирным. Она гарантирует, что из БД будут прочитаны только объекты, которые соответствуют поисковому термину. Из трех объектов, созданных с использованием данных в табл. 20.3, поисковому термину соответствуют только Alice и Peter, давая результаты, приведенные на рис. 20.5.

Рис. 20.5. Применение поискового термина в запросе

Избегание ловушки, связанной с оценкой метода `Like()`

Необходимо следить за тем, чтобы метод `EF.Functions.Like()` применялся только к объектам реализации `IQueryable<T>`. Следует избегать вызова метода `Like()` на объекте реализации `IEnumerable<T>` вроде показанного ниже:

```
...
public IActionResult Index(string searchTerm) {
  IEnumerable<Employee> data = context.Employees;
  if (!string.IsNullOrEmpty(searchTerm)) {
    data = data.Where(e => EF.Functions.Like(e.FirstName, searchTerm));
  }
  return View(data);
}
...
```

Результаты будут выглядеть такими же, но если изучить запрос, отправленный серверу баз данных, то несложно заметить, что ключевое слово `LIKE` в нем отсутствует:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[Salary],
[e].[SoftDeleted]
FROM [Employees] AS [e]
WHERE [e].[SoftDeleted] = 0
...
```

Инфраструктура Entity Framework Core извлечет все объекты, которые могли бы соответствовать поисковому термину, обработает их в приложении и отбросит те, которые не нужны. В примере приложения это означает, что запрос загрузит всего один добавочный объект, но в реальном приложении объем данных, которые загружаются и затем отбрасываются, может оказаться значительным.

Выполнение асинхронных запросов

Большинство запросов, делающихся с использованием Entity Framework Core, являются синхронными. В большинстве приложений синхронные запросы вполне приемлемы, поскольку запрос — единственная активность, которую проявляет метод действия ASP.NET Core MVC, обычно также синхронный.

Инфраструктура Entity Framework Core также способна выполнять запросы асинхронно, что может быть полезно, если вы применяете асинхронный метод действия *и* этому методу действия необходимо выполнять множество запросов одновременно, *плюс* только одной из таких активностей оказывается запрос к БД. Набор обстоятельств, в которых асинхронные запросы полезны, фактически настолько специфичен, что в большинстве проектов приложений ASP.NET Core MVC потребность в их использовании не возникает.

Перепишите метод действия `Index()`, чтобы сделать его асинхронным и применить в своих интересах поддержку асинхронных запросов, предлагаемую Entity Framework Core (листинг 20.16).

Листинг 20.16. Выполнение асинхронного запроса в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;
using System.Net.Http;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public async Task<IActionResult> Index(string searchTerm) {
      IQueryable<Employee> employees = context.Employees;
      if (!string.IsNullOrEmpty(searchTerm)) {
        employees = employees.Where(e =>
          EF.Functions.Like(e.FirstName, searchTerm));
      }
      HttpClient client = new HttpClient();
      ViewBag.PageSize = (await client.GetAsync("http://apress.com"))
        .Content.Headers.ContentLength;
      return View(await employees.ToListAsync());
    }

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

Ограничения асинхронных запросов затрудняют создание сколько-нибудь полезного примера. В листинге 20.16 с помощью класса `HttpClient` веб-сайту `apress.com` отправляется асинхронный HTTP-запрос `GET` с одновременным запрашиванием БД.

Избегание ловушки, связанной с параллельными запросами

В Microsoft специально предостерегают от использования объекта контекста для выполнения множества асинхронных запросов, поскольку класс `DbContext` не был написан с учетом приспособленности к ним. Это приводит к тому, что предприимчивые разработчики применяют внедрение зависимостей для получения двух объектов контекста, каждый из которых используется для выполнения параллельного асинхронного запроса:

```
...
public HomeController(AdvancedContext ctx, AdvancedContext ctx2) {
...
```

Проблема с таким подходом заключается в том, что средство внедрения зависимостей ASP.NET Core MVC будет создавать только один объект контекста и применять его для распознавания обеих зависимостей, а это означает наличие, в конечном счете, одного объекта контекста. Мой совет — принять ограничения поддержки асинхронных запросов.

Инфраструктура Entity Framework Core предлагает набор методов, которые навязывают асинхронную оценку запроса. Наиболее часто используемые асинхронные методы описаны в табл. 20.7, но асинхронные эквиваленты существуют для всех методов, которые навязывают асинхронную оценку, так что метод `LastAsync()` является асинхронным эквивалентом метода `Last()`. Асинхронные версии методов, применяемых для создания запроса, таких как `Where()`, не требуются, потому что они строят запрос, не выполняя его.

Таблица 20.7. Распространенные методы, которые выполняют асинхронный запрос

Имя	Описание
`LoadAsync()`	Навязывает асинхронное выполнение запроса, но ничего не делает с результатами. Является эквивалентом метода `Load()` и чаще всего используется с процессом исправления
`ToListAsync()`	Запрашивает БД и возвращает результирующие объекты в списке
`ToArrayAsync()`	Запрашивает БД и возвращает результирующие объекты в массиве
`ToDictionaryAsync(ключ)`	Запрашивает БД и возвращает результирующие объекты в словаре, применяя указанное свойство (`ключ`) как источник значений ключей
`CountAsync()`	Возвращает количество объектов, хранящихся в БД, которые соответствуют указанному предикату. Если предикат не указан, тогда возвращается количество хранящихся в БД объектов, соответствующих запросу
`FirstAsync(предикат)`	Возвращает первый объект, который соответствует указанному предикату (`предикат`)
`ForEachAsync(функция)`	Вызывает указанную функцию (`функция`) для каждого объекта, соответствующего запросу

На заметку! Метод `ForEachAsync()` не имеет синхронного эквивалента, но может использоваться для вызова функции для каждого объекта, который создается из результатов запроса.

В листинге 20.16 с помощью метода `ToListAsync()` производится асинхронный запрос БД и полученный список `List<Employee>` передается методу `View()`. Класс `List<T>` реализует интерфейс `IEnumerable<T>`, поэтому существующее представление способно выполнять перечисление объектов безо всяких изменений. Чтобы отобразить количество байтов, прочитанных из веб-сайта `apress.com` асинхронным HTTP-запросом в листинге 20.16, добавьте в представление `Index` элемент, выделенный полужирным в листинге 20.17.

Листинг 20.17. Добавление элемента в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <!-- ...для краткости содержимое таблицы не показано... -->
</table>
@if (ViewBag.PageSize != null) {
  <h4 class="bg-info p-2 text-center text-white">
    Page Size: @ViewBag.PageSize bytes
  </h4>
}
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Для тестирования асинхронного запроса запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`. Вы увидите элемент с размером страницы наряду с данными `Employee`, которые одновременно были извлечены из БД (рис. 20.6). (Точный размер страницы может варьироваться, т.к. веб-сайт `apress.com` часто обновляется.)

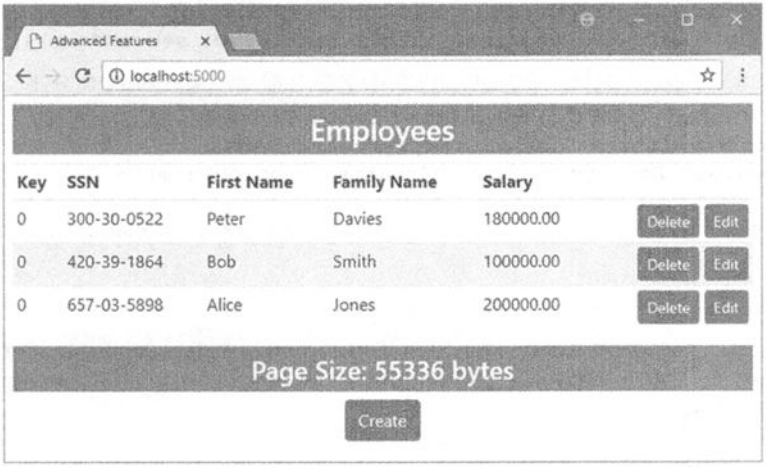

Рис. 20.6. Выполнение параллельных запросов

Явная компиляция запросов

Одной из наиболее привлекательных особенностей Entity Framework Core является способ, которым запросы LINQ транслируются в SQL. Процесс трансляции может быть сложным. Для повышения производительности инфраструктура Entity

Framework Core автоматически поддерживает кеш обработанных запросов и создает хешированное представление каждого запроса, который она обрабатывает, чтобы определить, доступна ли кешированная трансляция. Если кешированная трансляция доступна, то она и используется, а если нет, тогда создается новая трансляция, которая помещается в кеш для будущего применения.

Увеличить производительность такого процесса можно за счет явной трансляции запроса, чтобы инфраструктуре Entity Framework Core не пришлось создавать хеш-код и проверять кеш. Прием называется *явной компиляцией* запроса. Обновите контроллер `Home` для явной компиляции запроса, выполняемого действием `Index` (листинг 20.18).

Листинг 20.18. Компиляция запроса в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;
using System.Net.Http;
using System;
using System.Collections.Generic;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    private static Func<AdvancedContext, string, IEnumerable<Employee>> query
      = EF.CompileQuery((AdvancedContext context, string searchTerm)
        => context.Employees
          .Where(e => EF.Functions.Like(e.FirstName, searchTerm)));

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index(string searchTerm) {
      return View(string.IsNullOrEmpty(searchTerm )
        ? context.Employees : query(context, searchTerm));
    }
    // ...для краткости остальные методы действий не показаны...
  }
}
```

Операторы, которые производят скомпилированный запрос, могут оказаться трудными для восприятия. Компиляция выполняется с использованием метода `EF.CompileQuery()`:

```
...
EF.CompileQuery((AdvancedContext context, string searchTerm)
  => context.Employees.Where(e => EF.Functions.Like(e.FirstName,
                                                    searchTerm)));
...
```

В качестве аргумента методу `CompileQuery()` передается лямбда-выражение, которое принимает объект контекста и параметры, применяемые в запросе, и возвращает объект реализации `IQueryable<T>`. В листинге 20.18 лямбда-выражение получает объекты `AdvancedContext` и `string`, которые использует для создания

объекта реализации IQueryable<Employee>, запрашивающего БД с применением конструкции LIKE.

Результатом метода EF.CompileQuery() является объект Func<AdvancedContext, string, IEnumerable<Employee>>, представляющий функцию, которая принимает контекст и строку и производит последовательность объектов Employee:

```
...
private static Func<AdvancedContext, string, IEnumerable<Employee>> query
  = EF.CompileQuery((AdvancedContext context, string searchTerm)
    => context.Employees.Where(e => EF.Functions.Like(e.FirstName,
                                                      searchTerm)));
...
```

Обратите внимание, что скомпилированная функция возвращает объект реализации IEnumerable<T>, т.е. любые дальнейшие операции, предпринимаемые в отношении результата, будут выполняться в памяти, а не на основе запроса, который отправляется БД. Это имеет смысл, поскольку цель процесса — создание неизменяемого запроса, что означает необходимость включения в выражение, передаваемое методу CompileQuery(), всех требующихся аспектов запроса. Запрос выполняется путем вызова функции, возвращенной методом CompiledQuery():

```
...
return View(string.IsNullOrEmpty(searchTerm )
  ? context.Employees : query(context, searchTerm));
...
```

Никаких видимых отличий в способе выполнения скомпилированного запроса наблюдаться не будет, но "за кулисами" инфраструктура Entity Framework Core получает возможность пропустить процесс создания хешированного представления запроса и проверки, не транслировался ли он ранее.

Избегание ловушки, связанной с чрезмерными запросами

Обратите внимание, что параметр searchTerm для метода действия Index() проверяется на равенство null вне явно скомпилированного запроса. Распространенным заблуждением при определении выражения запроса является включение проверок, которые предназначены для выполнения в приложении, например:

```
...
EF.CompileQuery((AdvancedContext context, string searchTerm)
  => context.Employees.Where(e => string.IsNullOrEmpty(searchTerm)
    || EF.Functions.Like(e.FirstName, searchTerm)));
...
```

Проблема в том, что инфраструктура Entity Framework Core встроит проверку на предмет значений null в SQL-запрос, и это может оказаться не тем, что планировалось:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[Salary],
  [e].[SoftDeleted]
FROM [Employees] AS [e]
WHERE ([e].[SoftDeleted] = 0) AND ((@__searchTerm IS NULL
  OR (@__searchTerm = N'')) OR [e].[FirstName] LIKE @__searchTerm)
...
```

Такая ловушка противоположна ловушке, связанной с оценкой на стороне клиента, которая объясняется в следующем разделе, но обе проблемы подчеркивают важность исследования SQL-запросов, выпускаемых инфраструктурой Entity Framework Core, чтобы удостовериться в том, что они нацелены в точности на запланированные данные.

Избегание ловушки, связанной с оценкой на стороне клиента

При работе с Entity Framework Core может потребоваться некоторое время, прежде чем вы будете уверены в том, что запросы LINQ транслируются в желаемый код SQL. Как показано в книге, существует множество потенциальных ловушек, которые могут привести к тому, что из БД извлекается чересчур много или слишком мало данных. Одна потенциальная ошибка настолько распространена, что Entity Framework Core будет выдавать предупреждение при трансляции запроса в код SQL.

Проблема возникает, когда инфраструктура Entity Framework Core неспособна увидеть все детали запроса LINQ и не может полностью транслировать его в SQL. Это часто происходит, когда запрос подвергается рефакторингу, чтобы код, который выбирает набор объектов данных, мог использоваться более согласованно во всем приложении. Инфраструктура Entity Framework Core разделяет запрос с целью выполнения одной его части сервером баз данных, а другой части — клиентским приложением. В итоге не только увеличивается объем обработки, которую должно выполнить приложение, но может также значительно вырасти объем данных, извлекаемых запросом из БД.

В целях демонстрации добавьте в папку `Controllers` файл класса по имени `QueryController.cs` и определите в нем контроллер, как показано в листинге 20.19.

Листинг 20.19. Содержимое файла `QueryController.cs` из папки `Controllers`

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

namespace AdvancedApp.Controllers {

  public class QueryController : Controller {
    private AdvancedContext context;

    public QueryController(AdvancedContext ctx) => context = ctx;

    public IActionResult ServerEval() {
      return View("Query", context.Employees.Where(e => e.Salary > 150_000));
    }

    public IActionResult ClientEval() {
      return View("Query", context.Employees.Where(e => IsHighEarner(e)));
    }
    private bool IsHighEarner(Employee e) {
      return e.Salary > 150_000;
    }
  }
}
```

В контроллере определены два действия, запрашивающие объекты `Employee`, у которых значение свойства `Salary` превышает 150 000. Метод `ServerEval()` помещает выражение фильтра прямо в конструкцию `Where()` выражения LINQ, тогда как метод `ClientEval()` применяет отдельный метод, что представляет собой типичный процесс рефакторинга, который позволяет вынести критерий для выбора служащих, получающих высокую зарплату, в отдельный метод.

Чтобы снабдить оба действия представлением, создайте папку `Views/Query` и добавьте в нее файл по имени `Query.cshtml` с содержимым из листинга 20.20.

Листинг 20.20. Содержимое файла `Query.cshtml` из папки `Views/Query`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="4" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
      </tr>
    }
  </tbody>
</table>
```

Для выяснения отличий в способе выполнения запросов запустите приложение, используя `dotnet run`, перейдите по ссылкам `http://localhost:5000/query/servereval` и `http://localhost:5000/query/clienteval`; результаты обоих запросов показаны на рис. 20.7.

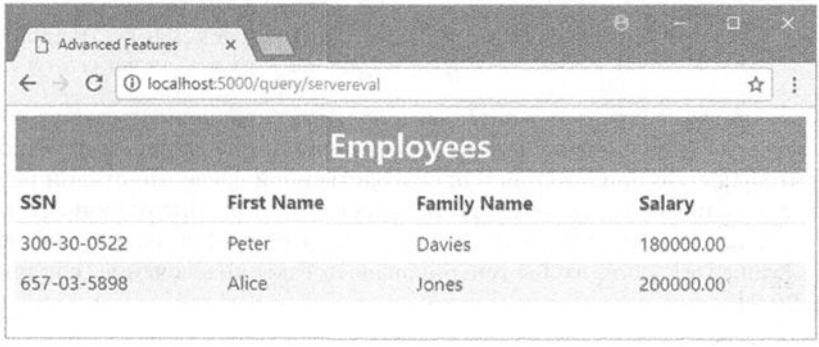

Рис. 20.7. Результаты запросов

Чтобы понять разницу между двумя методами действий, потребуется исследовать запросы, отправленные серверу баз данных. Метод действия `ServerEval()` дает такой запрос:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[Salary],
[e].[SoftDeleted]
FROM [Employees] AS [e]
WHERE ([e].[SoftDeleted] = 0) AND ([e].[Salary] > 150000.0)
...
```

Конструкция `WHERE` в запросе приведет к извлечению только объектов `Employee` со значением свойства `Salary`, превышающим 150 000. Таким образом, из БД будут извлечены только объекты, которые впоследствии отобразятся пользователю.

А вот запрос, который выпускается методом действия `ClientEval()`:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[Salary],
[e].[SoftDeleted]
FROM [Employees] AS [e]
WHERE [e].[SoftDeleted] = 0
...
```

Инфраструктура Entity Framework Core не в состоянии просмотреть метод `IsHighEarner()` контроллера и встроить содержащуюся внутри него логику в запрос SQL. Взамен инфраструктура Entity Framework Core транслирует ту часть запроса, которую видит, в запрос SQL и прогоняет получаемые объекты через метод `IsHighEarner()` для построения результата запроса. Невыбранные методом `IsHighEarner()` объекты отбрасываются, что в итоге приводит к чтению большего объема данных из БД и увеличению объема работы, которую приходится выполнять приложению для получения требуемых результатов. В примере приложения это означает чтение и создание одного добавочного объекта, но в реальном приложении разница может оказаться существенной.

Генерация исключения, связанного с оценкой на стороне клиента

Когда часть запроса должна оцениваться на стороне клиента, инфраструктура Entity Framework Core отобразит в журнальном выводе предупреждающее сообщение следующего вида:

```
The LINQ expression 'where value(AdvancedApp.Controllers.QueryController)
.IsHighEarner([e])' could not be translated and will be evaluated locally
```

Выражение LINQ 'where value(AdvancedApp.Controllers.QueryController) .IsHighEarner([e])' не может быть транслировано и будет оцениваться локально

Его довольно легко упустить в потоке журнальных сообщений, и оценка на стороне клиента может остаться незамеченной до тех пор, пока не обнаружатся проблемы с производительностью приложения в производственной среде. На стадии разработки удобно получать исключение, когда часть запроса будет оцениваться на стороне клиента, что сделает проблему более заметной. Измените конфигурацию инфраструктуры Entity Framework Core, чтобы генерировалось такое исключение, как показано в листинге 20.21.

Листинг 20.21. Конфигурирование исключения в файле `Startup.cs` из папки `AdvancedApp`

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

namespace AdvancedApp {
  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<AdvancedContext>(options =>
        options.UseSqlServer(conString).ConfigureWarnings(warning =>
          warning.Throw(RelationalEventId.QueryClientEvaluationWarning)));
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
      app.UseDeveloperExceptionPage();
      app.UseStatusCodePages();
      app.UseStaticFiles();
      app.UseMvcWithDefaultRoute();
    }
  }
}
```

Метод `ConfigureWarnings()` применяется для конфигурирования предупреждений, выдаваемых инфраструктурой Entity Framework Core, с использованием лямбда-выражения, принимающего объект `WarningsConfigurationsBuilder`, который определяет методы, описанные в табл. 20.8.

Таблица 20.8. Методы `WarningsConfigurationBuilder`

Имя	Описание
`Ignore(событие)`	Сообщает инфраструктуре Entity Framework Core о необходимости игнорирования указанного события
`Log(событие)`	Сообщает инфраструктуре Entity Framework Core о необходимости занесения в журнал указанного события
`Throw(событие)`	Сообщает инфраструктуре Entity Framework Core о необходимости генерации исключения для указанного события

Методы в табл. 20.8 применяются с перечислением `RelationalEventId`, определяющим значения, которые представляют диагностические события, вероятно встречающиеся в приложении Entity Framework Core. Предусмотрены значения для почти трех десятков разных событий, хотя большинство из них относятся к жизненным циклам подключений к БД и транзакций, а также к созданию и применению миграций.

Полный список событий доступен по ссылке

```
https://docs.microsoft.com/ru-ru/ef/core/api/
  microsoft.entityframeworkcore.infrastructure.relationaleventid
```

В листинге 20.20 использовалось значение `QueryClientEvaluationWarning`, которое представляет событие, инициируемое инфраструктурой Entity Framework Core, когда часть запроса будет оцениваться на стороне клиента. Чтобы увидеть эффект от изменения, запустите приложение с помощью `dotnet run` и перейдите по ссылке `http://localhost:5000/query/clienteval`. Вместо легко упускаемого предупреждения вы получите исключение (рис. 20.8).

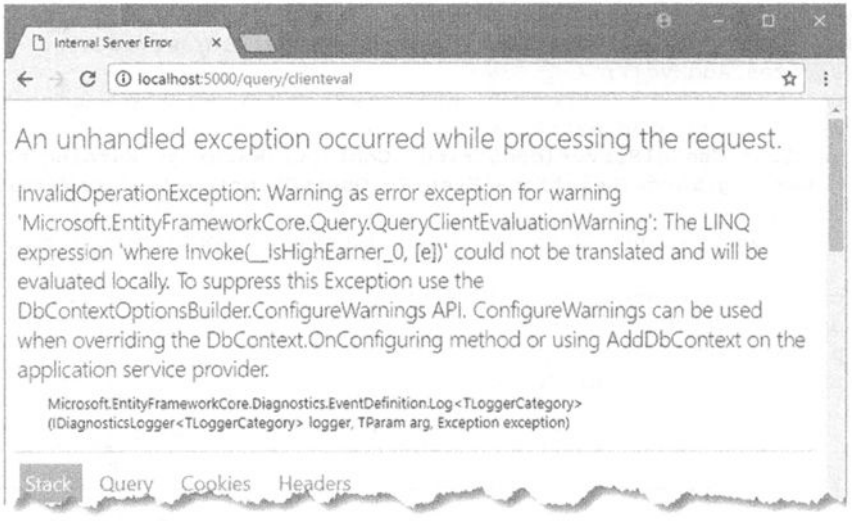

Рис. 20.8. Исключение, связанное с оценкой части запроса на стороне клиента

Резюме

В главе были описаны расширенные возможности, которые инфраструктура Entity Framework Core предлагает при запрашивании данных. Приводились объяснения, как управлять средством отслеживания изменений, каким образом применять фильтр запросов, как выполнять запросы с использованием SQL-выражения `LIKE` и каким образом явно компилировать запросы. В заключение была продемонстрирована распространенная проблема, возникающая из-за рефакторинга запросов, и показано, как сконфигурировать приложение, чтобы знать о присутствии этой проблемы в собственных проектах. В следующей главе рассматриваются расширенные возможности для хранения данных.

ГЛАВА 21

Хранение данных

В этой главе будет продолжено описание расширенных возможностей, которые предоставляет инфраструктура Entity Framework Core, с акцентированием внимания на возможностях, относящихся к добавлению или обновлению данных. Будет показано, как выбирать типы данных, используемые для хранения значений, каким образом скрывать значения данных от остальных частей приложения и как обнаруживать параллельные обновления, выполняемые множеством клиентов. В табл. 21.1 приведены сведения, позволяющие поместить расширенные функции хранения данных в контекст.

Таблица 21.1. Помещение расширенных функций хранения данных в контекст

Вопрос	Ответ
Что это такое?	Расширенные функции хранения данных позволяют изменить способ хранения данных в БД, переопределяя традиционное поведение. Изменения могут распространяться от выбора специфических типов SQL до обнаружения ситуации, когда два пользователя обновляют те же самые данные
Чем они полезны?	Расширенные функции хранения данных могут быть полезны при моделировании существующей БД или при наличии у приложения специфических потребностей, которые нелегко удовлетворить с помощью стандартных средств
Как они используются?	Расширенные функции хранения данных применяются через комбинацию членов классов модели данных и операторов Fluent API
Существуют ли какие-то скрытые ловушки или ограничения?	Расширенные функции хранения данных изменяют способ отображения данных на объекты модели данных, что может привести к странным результатам в случае невнимательного использования
Существуют ли альтернативы?	В большинстве проектов можно применять стандартные возможности Entity Framework Core для хранения данных

В табл. 21.2 приведена сводка по главе.

Таблица 21.2. Сводка по главе

Задача	Решение	Листинг
Изменение типа данных SQL, используемого для представления значения	Применяйте метод `HasColumnType()` или `HasMaxLength()`	21.1–21.6
Обработка значений перед тем, как они делаются доступными остальным частям приложения или сохраняются в БД	Используйте поддерживающее поле	21.7–21.11
Сокрытие значений данных для части MVC приложения	Применяйте теневое свойство	21.12–21.15
Установка стандартного значения	Используйте метод `HasDefaultValue()`	21.16–21.20
Обнаружение параллельных обновлений	Применяйте маркер параллелизма или включите ведение версий строк	21.21–21.30

Подготовительные шаги

В главе продолжается работа с проектом `AdvancedApp`, который был создан в главе 19 и модифицирован в главе 20. В качестве подготовки измените код в контроллере `Home`, чтобы включить связанные данные в запрос, выполняемый методом действия `Index()`, и привести в порядок метод `Update()` с целью обновления всех свойств (листинг 21.1).

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 21.1. Упрощение кода в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(context.Employees.Include(e => e.OtherIdentity));
    }
    public IActionResult Edit(string SSN, string firstName, string familyName)
    {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
```

```
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
      } else {
        context.Update(employee);
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Attach(employee);
      employee.SoftDeleted = true;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы отобразить пользователю детали объектов `SecondaryIdentity`, внесите изменения, приведенные в листинге 21.2, в представление `Index.cshtml` из папки `Views/Home`. Также воспользуйтесь возможностью и удалите содержимое, добавленное в предыдущей главе, которое больше не требуется.

Листинг 21.2. Добавление содержимого в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th>In Use</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="7" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
```

```
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td>@(e.OtherIdentity?.InActiveUse.ToString() ?? "(N/A)")</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName"
                  value="@e.FamilyName" />
            <button type="submit" asp-action="Delete" formmethod="post"
                  class="btn btn-sm btn-danger">Delete</button>
            <button type="submit" asp-action="Edit" formmethod="get"
                  class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Выполните в папке проекта AdvancedApp команды из листинга 21.3 для удаления и воссоздания БД.

Листинг 21.3. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением dotnet run, перейдите по ссылке http://localhost:5000, щелкните на кнопке Create (Создать) и сохраните в БД три объекта Employee, используя значения из табл. 21.3.

Таблица 21.3. Значения данных для создания объектов в примере

SSN (Номер карточки социального страхования)	First Name (Имя)	Family Name (Фамилия)	Salary (Оклад)	Other Name (Другое имя)	In Active Use (Активно используется)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
657-03-5898	Alice	Jones	200000	Allie	Флажок отмечен
300-30-0522	Peter	Davies	180000	Pete	Флажок отмечен

Результат создания трех объектов Employee показан на рис. 21.1.

Рис. 21.1. Выполнение примера приложения

Указание типов данных SQL

Инфраструктура Entity Framework Core автоматически позаботится об отображении между типами данных .NET и SQL, как в миграции для проекта "сначала код", так и на стадии формирования шаблонов проекта "сначала БД". Серверы баз данных не всегда поддерживают те же самые типы данных — или реализуют их одинаково — и за выбор подходящих типов несет ответственность поставщик БД. Это означает, например, что миграции для той же самой модели данных, которые нацелены на два разных сервера баз данных, могут применять отличающиеся типы SQL.

Большинство поставщиков используют похожие типы, но есть некоторые вариации. В табл. 21.4 представлены отображения, применяемые для элементарных типов .NET Core в случае официального поставщика баз данных SQL Server и одного из самых популярных поставщиков MySQL.

Таблица 21.4. Отображения, применяемые поставщиками баз данных для элементарных типов .NET Core

Тип .NET Core	Тип SQL Server	Тип MySQL
`int`	`int`	`int`
`long`	`bigint`	`bigint`
`bool`	`bit`	`bit`
`byte`	`tinyint`	`tinyint`
`double`	`float`	`double`
`char`	`int`	`tinyint`
`short`	`smallint`	`smallint`
`float`	`real`	`float`
`decimal`	`decimal(18,2)`	`decimal(65,30)`
`string`	`nvarchar(max)`	`longtext`
`TimeSpan`	`time`	`time(6)`
`DateTime`	`datetime2`	`datetime(6)`
`DateTimeOffset`	`datetimeoffset`	`datetime(6)`
`Guid`	`uniqueidentifier`	`char(36)`

Внимание! В будущих выпусках поставщиков баз данных отображения могут измениться. Для окончательного определения, какой тип данных использует поставщик, создайте миграцию.

Вы можете изменить тип данных SQL, используемый для хранения значения, если выбранный инфраструктурой Entity Framework Core тип не подходит. Такая возможность чаще всего применяется при подгонке типа, чтобы обеспечить достаточную точность для значений, генерируемых приложением, или при выборе меньшего типа данных для ограничения значений, которые можно хранить.

Свойство `Salary` класса `Employee` выражается как .NET-тип `decimal`, который поставщик SQL Server отображает на SQL-тип `decimal(18,2)` — число с 18 цифрами слева от десятичной точки и 2 цифрами справа. Такой тип точнее, чем необходимо для выражения зарплаты особы, а потому сообщите инфраструктуре Entity Framework Core о том, что нужно использовать менее точный тип, переопределив стандартный тип, который был выбран при создании первой миграции в главе 19 (листинг 21.4).

Внимание! Инфраструктура Entity Framework Core не проверяет типы или указываемые максимальные длины, т.е. вы должны быть уверены в том, что типы и размеры подходят для имеющихся потребностей.

Листинг 21.4. Изменение типа данных в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)");

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
    }
  }
}
```

Метод `HasColumnType()` позволяет указать тип SQL для свойства, которое было выбрано с использованием метода `Property()`. В листинге 21.4 указан тип `decimal(8,2)`, сокращающий количество цифр слева от десятичной точки до восьми.

На заметку! Если вам не нравится работать с Fluent API, тогда можете указать тип SQL для свойства с применением атрибута `Column`, предоставив в нем аргумент `TypeName`: `[Column(TypeName = "decimal(8, 2)")]`.

Указание максимальной длины

В случае работы со значением, которое будет сохраняться в БД с использованием типа данных в форме массива, такого как `string` либо `int[]`, можете предоставить инфраструктуре Entity Framework Core инструкцию относительно объема данных, которые нужно хранить, не выбирая явно тип данных SQL. Таким образом, вы можете влиять на выбор типа данных без необходимости в принятии решений, специфичных для конкретного поставщика БД или сервера. Установите максимальную длину для свойства `Name`, определенного в классе `SecondaryIdentity`, как показано в листинге 21.5.

На заметку! Если вы предпочитаете не работать с Fluent API, тогда можете указать максимальную длину, декорируя свойство атрибутом `MaxLength`.

Листинг 21.5. Установка максимальной длины в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)");

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
```

```
        modelBuilder.Entity<SecondaryIdentity>()
          .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Для указания максимальной длины применяется метод `HasMaxLength()` и для свойства `Name` было задано максимум 100 символов.

Обновление базы данных

Изменение типа данных или указание максимальной длины требует миграции для обновления БД. Выполните в папке проекта `AdvancedApp` команды из листинга 21.6, чтобы создать новую миграцию и применить ее к БД.

Листинг 21.6. Создание и применение миграции к БД

```
dotnet ef migrations add ChangeType
dotnet ef database update
```

Если вы просмотрите метод `Up()` в папке `<отметка времени>_ChangeType.cs` в папке `Migrations`, то увидите эффект от методов `HasColumnType()` и `HasMaxLength()`:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.AlterColumn<string>(name: "Name",
                                       table: "SecondaryIdentity",
    maxLength: 100, nullable: true, oldClrType: typeof(string),
    oldNullable: true);

  migrationBuilder.AlterColumn<decimal>(name: "Salary",table: "Employees",
    type: "decimal(8,2)",nullable: false, oldClrType: typeof(decimal));
}
...
```

Тип данных, указанный в листинге 21.4, вмещает числа, которые имеют восемь цифр слева от десятичной точки. Для выяснения, что произойдет, если число превысит доступное хранилище, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Edit (Редактировать) для одного из элементов. Измените значение в поле Salary (Оклад) на `100000000` и щелкните на кнопке Save (Сохранить). Инфраструктура Entity Framework Core попытается обновить БД, но из-за того, что значение для оклада содержит больше цифр, чем способен хранить указанный тип данных, она сообщит об ошибке (рис. 21.2). Конкретная ошибка будет зависеть от природы несоответствия, но суть в том, что вы должны обеспечить, чтобы приложение не пыталось и не позволяло пользователю сохранять значения, которые не могут быть представлены БД.

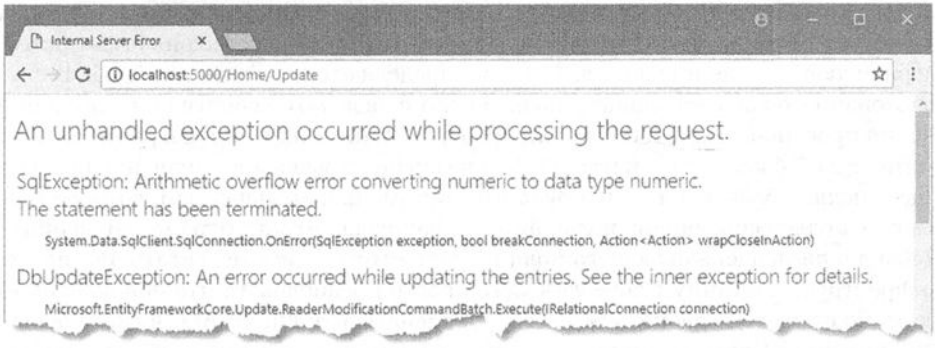

Рис. 21.2. Превышение доступного хранилища для значения данных

Проверка достоверности или форматирование значений данных

Во многих приложениях сущностные классы являются просто коллекциями свойств, которые обеспечивают удобный доступ к данным в БД. Однако в ряде ситуаций предоставление прямого доступа к значениям данных сопряжено с проблемами, т.к. требуется какой-то вид обработки или проверки достоверности.

В Entity Framework Core предусмотрены *поддерживающие поля*, которые хранят значения в БД, но не доступны остальным частям приложения. Другие части приложения взамен получают доступ к данным опосредовано через свойства. Понять концепцию поддерживающих полей легче посредством демонстрации, поэтому модифицируйте свойство `Salary` в классе `Employee` так, чтобы оно больше не давало прямого доступа к значению в БД, а применяло поддерживающее поле (листинг 21.7).

Листинг 21.7. Определение поддерживающего поля в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary * 2;
      set => databaseSalary = Math.Max(0, value);
    }

    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;
  }
}
```

Поддерживающее поле называется databaseSalary. Оно должно иметь тип, совместимый со свойством, с которым нужно работать, и в рассматриваемом примере поддерживающее поле и свойство Salary определяются с типом decimal. В случае использования поддерживающего поля можно переделать свойство Salary для выполнения проверки достоверности или преобразования значения, сохраняемого в БД. В листинге 21.7 блок get свойства Salary возвращает удвоенное значение поддерживающего поля, а блок set гарантирует, что минимальным значением, которое можно присвоить поддерживающему полю, будет 0, запрещая отрицательные значения.

Добавления поддерживающего поля в сущностный класс недостаточно, поскольку инфраструктура Entity Framework Core просто предположит, что она должна продолжать пользоваться свойством Salary. Задействуйте Fluent API, чтобы сообщить Entity Framework Core о наличии поддерживающего поля и о том, как с ним работать (листинг 21.8).

На заметку! Поддерживающие поля могут конфигурироваться только с применением Fluent API. Поддержка со стороны атрибутов для поддерживающих полей отсутствует.

Листинг 21.8. Настройка поддерживающего поля в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) {}
    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });

      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Имеется возможность дальнейшего конфигурирования свойства за счет соединения вызовов с существующим оператором Fluent API, выбирающим свойство `Salary`, который использовался в предыдущем разделе для изменения типа данных. Поддерживающее поле настраивается с применением метода `HasField()`, которому в аргументе указывается имя свойства. Способ использования поддерживающего поля конфигурируется вызовом метода `UsePropertyAccessMode()`, которому передается значение перечисления `PropertyAccessMode`, описанного в табл. 21.5.

Совет. Применять метод `UsePropertyAccessMode()` для конфигурирования поддерживающего поля не обязательно, но его вызов гарантирует использование поля инфраструктурой Entity Framework Core ожидаемым способом и делает целевое назначение поддерживающего поля очевидным для других разработчиков, читающих операторы Fluent API.

Таблица 21.5. Значения `PropertyAccessMode`

Имя	Описание
`FieldDuringConstruction`	Значение по умолчанию, которое сообщает инфраструктуре Entity Framework Core о необходимости применения поддерживающего поля при первоначальном создании объекта и затем использования свойства для всех остальных операций, включая обнаружение изменений
`Field`	Это значение заставляет инфраструктуру Entity Framework Core игнорировать свойство и всегда применять поддерживающее поле
`Property`	Это значение заставляет инфраструктуру Entity Framework Core всегда использовать свойство и игнорировать поддерживающее поле

Выбор правильного значения `PropertyAccessMode` важен, т.к. выбранное значение заставляет инфраструктуру Entity Framework Core вести себя по-разному. В листинге 21.8 поддерживающее поле всегда имеет "подлинное" значение, а потому применяется значение `Field`, которое гарантирует, что Entity Framework Core будет присваивать значение из БД поддерживающему полю при создании объекта на основе данных запроса и использовать значение поддерживающего поля при записывании обновлений в БД.

В случае добавления поддерживающего поля никаких изменений вносить в БД не придется, потому что оно влияет только на то, как значения данных отображаются на сущностный класс. Чтобы увидеть эффект от изменений в настоящем разделе, запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`. Когда инфраструктура Entity Framework Core запрашивает данные для объектов `Employee`, она присваивает значение из столбца `Salary` поддерживающему полю. Когда представление Razor перечисляет объекты `Employee`, оно читает значение свойства `Salary`, чей блок `get` возвращает удвоенное значение, хранящееся в БД, и производит результат, показанный на рис. 21.3.

Чтобы посмотреть, как изменение значения свойства влияет на поддерживающее поле, щелкните на кнопке Edit для одного из объектов `Employee` и введите в поле Salary значение `120000`.

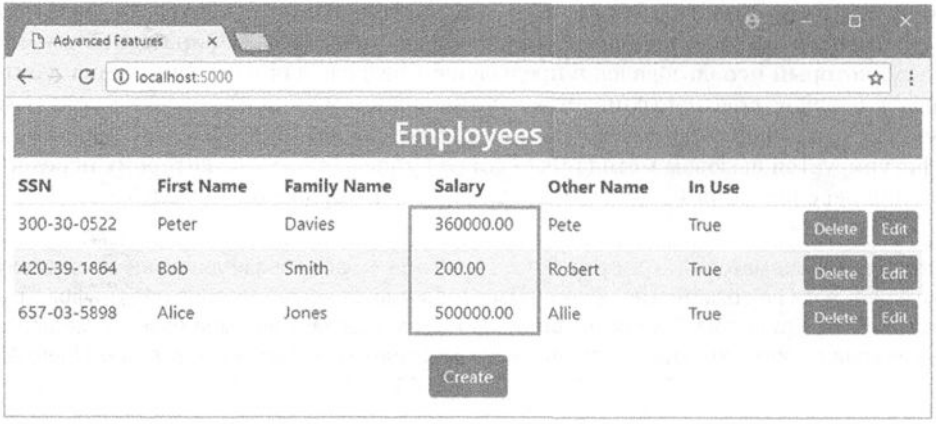

Рис. 21.3. Использование поддерживающего поля для опосредованного доступа к значениям

Поддерживающее поле сконфигурировано со значением `PropertyAccessMode.Field`, т.е. для обновления БД применяется значение поддерживающего поля, а не свойства. Когда инфраструктура Entity Framework Core производит обнаружение изменений, то именно значение поддерживающего поля, а не блок `get` свойства `Salary` используется при обновлении БД. Для подтверждения этого выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) в среде Visual Studio, подключитесь к БД и выполните запрос, приведенный в листинге 21.9.

Листинг 21.9. Запрашивание БД

```
USE AdvancedDb
SELECT * FROM Employees
```

Результаты запроса показывают, что при обновлении столбца `Salary` в таблице `Employees` инфраструктура Entity Framework Core применяла поддерживающее поле, как видно в табл. 21.6.

Таблица 21.6. Результаты запроса из листинга 21.9

FamilyName	FirstName	SSN	Salary
Davies	Peter	300-30-0522	180000
Jones	Alice	657-03-5898	200000
Smith	Bob	420-39-1864	120000

Если бы использовалось значение `FieldDuringConstruction`, тогда для получения значения при обновлении БД инфраструктура Entity Framework Core применяла бы блок `get` свойства `Salary`.

Остальные части приложения по-прежнему читают обновленное значение посредством блока `get` свойства `Salary`, оставаясь в неведении о поддерживающем поле. Это видно в представлении, отображаемом частью ASP.NET Core MVC приложения, где показано удвоенное значение, хранящееся в БД (рис. 21.4).

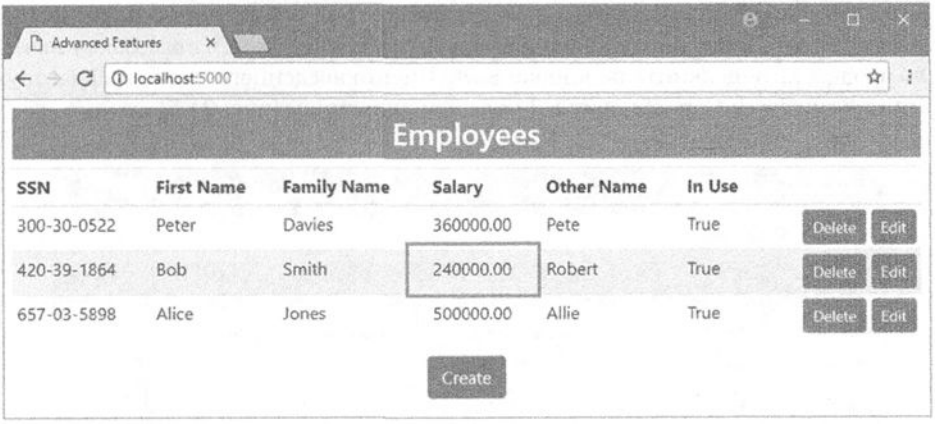

Рис. 21.4. Обновление значения с помощью поддерживающего поля

Избегание ловушки, связанной с выборочным обновлением поддерживающего поля

При реализации блоков `set`, которые не всегда обновляют поддерживающее поле, следует соблюдать осторожность. Чтобы выявить проблему, измените блок `set` для свойства `Salary` с целью обновления поддерживающего поля только в случае четных значений (листинг 21.10). Модифицируйте также блок `get` для свойства `Salary`, чтобы он возвращал неизмененное значение поддерживающего поля, что облегчит понимание проблемы.

Листинг 21.10. Выборочное обновление поддерживающего поля в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary;
      set {
        if (value % 2 == 0) {
          databaseSalary = value;
        }
      }
    }
    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;
  }
}
```

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на одной из кнопок Edit. Введите в поле Salary нечетное число вроде `101` и щелкните на кнопке Save. Вместо введенного числа инфраструктура Entity Framework Core обновит БД нулевым значением (рис. 21.5).

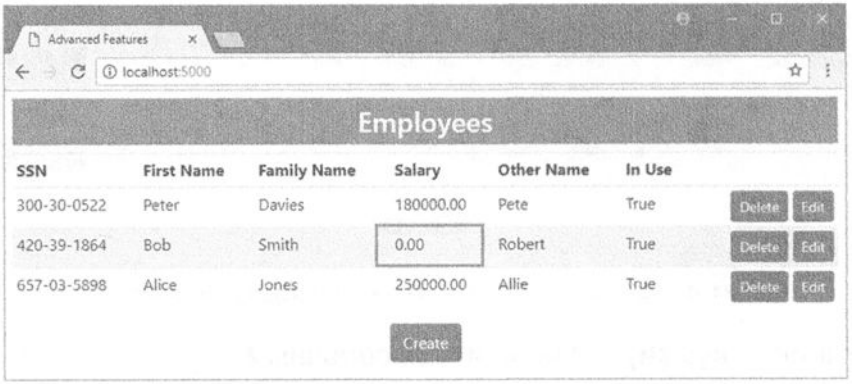

Рис. 21.5. Эффект от выборочного обновления поддерживающего поля

Одна лишь инфраструктура Entity Framework Core знает о поддерживающих полях и умеет обращаться с ними. В приложении ASP.NET Core MVC связыватель моделей MVC также несет ответственность за создание объектов. Когда браузер отправляет контроллеру HTTP-запрос `POST`, связыватель моделей создает новый объект `Employee` и устанавливает его свойства с применением значений, введенных пользователем. В отличие от ситуации, когда объект `Employee` создается инфраструктурой Entity Framework Core, связыватель моделей игнорирует поддерживающее поле и для присваивания значения из HTTP-запроса использует блок `set` свойства `Salary`. Но блок `set` игнорирует значение, присвоенное связывателем моделей MVC, потому что оно не является четным числом. Таким образом, при передаче объекта `Employee` объекту контекста, чтобы инфраструктура Entity Framework Core смогла выполнить обновление, поддерживающее поле имеет свое стандартное значение (нулевое в случае `decimal`).

Лучший способ избежать проблемы — реализовывать блоки `set`, которые всегда обновляют свои поддерживающие поля. Если это невозможно, тогда нужно запросить БД, в результате чего инфраструктура Entity Framework Core создаст объект с существующими значениями и применит непосредственно к нему значения из HTTP-запроса, как показано в листинге 21.11.

Листинг 21.11. Установка значений напрямую в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;
```

```
    public HomeController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Employees.Include(e => e.OtherIdentity));
    }
    public IActionResult Edit(string SSN, string firstName, string familyName) {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
    [HttpPost]
    public IActionResult Update(Employee employee, decimal salary) {
      Employee existing = context.Employees.Find(employee.SSN,
        employee.FirstName, employee.FamilyName);
      if (existing == null) {
        context.Add(employee);
      } else {
        existing.Salary = salary;
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Attach(employee);
      employee.SoftDeleted = true;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Метод `Update()` изменен так, чтобы запрашивать у БД существующий объект `Employee`, что дает объект, созданный Entity Framework Core, с корректно инициализированным поддерживающим полем. Затем присваивается значение, указанное пользователем в поле Salary, которое принимается за счет добавления параметра `salary` к методу действия:

```
...
public IActionResult Update(Employee employee, decimal salary) {
...
```

Такой прием предоставляет значение, введенное пользователем в HTML-форме, которое можно присвоить свойству `Salary`:

```
...
existing.Salary = salary;
...
```

Если блок `set` обновляет поддерживающее поле, тогда инфраструктура Entity Framework Core выявит измененное значение и обновит БД. Если блок `set` отбрасывает новое значение (поскольку оно нечетное), то инфраструктура Entity Framework Core обнаружит, что изменения отсутствуют, и первоначальное значение, хранящееся в БД, сбережется.

Запустите приложение, перейдите по ссылке `http://localhost:5000` и повторите ввод в поле Salary четного числа, такого как `100`. На этот раз при сохранении изменений нечетное число отбрасывается, но четное число по-прежнему будет храниться в БД (рис. 21.6).

Рис. 21.6. Избегание установки поддерживающего поля в стандартное значение для его типа

Сокрытие значений данных от части MVC приложения

Существуют типы данных, которые необходимы для выполнения операций с БД, но не должны быть доступными части MVC приложения либо из-за того, что данные являются конфиденциальными, либо потому, что приложение желательно сосредоточить на данных, непосредственно видимых пользователю. В таких ситуациях можно использовать *теневые свойства* — свойства, которые определены в модели данных, но не в сущностном классе, представляющем эти данные.

Самый распространенный сценарий применения теневых свойств связан с отслеживанием времени, в течение которого объекты хранились в БД; теневые свойства предоставляют информацию, полезную при диагностике проблем, но такие сведения пользователю не интересны и не должны отображаться частью ASP.NET Core MVC приложения.

Теневые свойства создаются с использованием Fluent API. Добавьте в класс `Employee` теневое свойство по имени `LastUpdated` (листинг 21.12).

На заметку! Теневые свойства могут конфигурироваться только с применением Fluent API. Поддержка со стороны атрибутов для теневых свойств отсутствует.

Листинг 21.12. Определение теневого свойства в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {
```

```
public class AdvancedContext : DbContext {
  public AdvancedContext(DbContextOptions<AdvancedContext> options)
    : base(options) {}
  public DbSet<Employee> Employees { get; set; }
  protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Employee>()
      .HasQueryFilter(e => !e.SoftDeleted);
    modelBuilder.Entity<Employee>().Ignore(e => e.Id);
    modelBuilder.Entity<Employee>()
      .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
    modelBuilder.Entity<Employee>()
      .Property(e => e.Salary).HasColumnType("decimal(8,2)")
      .HasField("databaseSalary")
      .UsePropertyAccessMode(PropertyAccessMode.Field);
    modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated");
    modelBuilder.Entity<SecondaryIdentity>()
      .HasOne(s => s.PrimaryIdentity)
      .WithOne(e => e.OtherIdentity)
      .HasPrincipalKey<Employee>(e => new { e.SSN,
        e.FirstName, e.FamilyName })
      .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
        s.PrimaryFirstName, s.PrimaryFamilyName });
    modelBuilder.Entity<SecondaryIdentity>()
      .Property(e => e.Name).HasMaxLength(100);
  }
}
```

Теневое свойство определяется с использованием метода `Entity()` для выбора класса и последующего вызова метода `Property()`. Это другая версия метода `Property()`, которая отличается от версии, применяемой в предшествующих примерах. В аргументе указывается имя теневого свойства, а в параметре типа — его тип данных; оператор Fluent API в листинге 21.12 сообщает инфраструктуре Entity Framework Core о том, что имеется свойство типа `DateTime` по имени `LastUpdated`.

Совет. Теневые свойства можно конфигурировать подобно любым другим свойствам, выстраивая цепочку обращений к методам вроде `IsRequired()`.

Добавление теневого свойства в существующую БД требует миграции. Выполните в папке проекта `AdvancedDb` команды из листинга 21.13, чтобы создать миграцию по имени `ShadowProperty` и применить ее к БД.

Листинг 21.13. Создание и применение миграции к БД

```
dotnet ef migrations add ShadowProperty
dotnet ef database update
```

Если вы просмотрите метод `Up()` в файле `<отметка времени>_ShadowProperty.cs`, который был создан внутри папки `Migrations`, то увидите, каким образом инфраструктура Entity Framework Core настроила столбец для теневого свойства, несмотря на отсутствие соответствующего свойства в классе `Employee`:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.AddColumn<DateTime>(
    name: "LastUpdated",
    table: "Employees",
    nullable: false,
    defaultValue: new DateTime(1, 1, 1, 0, 0, 0, 0,
                               DateTimeKind.Unspecified));
}
...
```

Доступ к значениям теневых свойств

Теневые свойства доступны через класс контекста. Модифицируйте метод действия `Update()` в контроллере `Home`, чтобы присваивать значение теневому свойству `LastUpdated`, когда в БД вносится изменение (листинг 21.14).

Листинг 21.14. Обновление теневого свойства в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(Employee employee, decimal salary) {
  Employee existing = context.Employees.Find(employee.SSN,
    employee.FirstName, employee.FamilyName);
  if (existing == null) {
    context.Entry(employee)
      .Property("LastUpdated").CurrentValue = System.DateTime.Now;
    context.Add(employee);
  } else {
    existing.Salary = salary;
    context.Entry(existing)
      .Property("LastUpdated").CurrentValue = System.DateTime.Now;
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Метод `Entry()` использовался в главе 12 для доступа к средству обнаружения изменений Entity Framework Core, но возвращаемый им объект также может применяться для доступа к теневым свойствам через его метод `Property()`. Значение теневого свойства можно прочитать или установить с использованием свойства `CurrentValue`: в листинге 21.14 теневое свойство устанавливается в текущее показание времени.

На заметку! Доступ к теневым свойствам возможен только через класс контекста, т.е. даже если работать с контекстом напрямую в контроллере, как делалось в примере, связыватель моделей MVC не сможет установить значение любых теневых свойств невзирая на их присутствие в HTTP-запросе.

Включение теневых свойств в запросы

Для включения теневых свойств в запросы LINQ применяется статический метод `EF.Property()`, что означает возможность встраивания теневых свойств в запросы. Воспользуйтесь методом `EF.Property()`, чтобы упорядочить объекты в БД по значению теневого свойства (листинг 21.15).

Листинг 21.15. Запрашивание с применением теневого свойства в файле HomeController.cs из папки Controllers

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(context.Employees.Include(e => e.OtherIdentity)
        .OrderByDescending(e => EF.Property<DateTime>(e, "LastUpdated")));
    }

    // ...для краткости остальные действия не показаны...
  }
}
```

Метод `EF.Property()` принимает запрашиваемый объект и имя теневого свойства. В параметре типа должен быть указан тип, который использовался для определения свойства в операторе Fluent API. Чтобы взглянуть на работу с теневым свойством, запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000`.

Отредактируйте второй или третий объект `Employee` и щелкните на кнопке Save; вы увидите, что он отобразится в первой строке таблицы (рис. 21.7).

Установка стандартных значений

При настройке свойства `LastUpdated` в предыдущем разделе установка значения производилась в двух ситуациях: при первом сохранении объекта в БД и при модификации существующего объекта.

Один из этих операторов можно удалить, поручив инфраструктуре Entity Framework Core установку значения для свойства `LastUpdated`, когда сохраняется новый объект. Установите стандартное значение для свойства `LastUpdated` с применением метода `HasDefaultValue()` в операторе Fluent API, как показано в листинге 21.16.

На заметку! Стандартные значения могут конфигурироваться только с использованием Fluent API. Поддержка со стороны атрибутов для стандартных значений отсутствует.

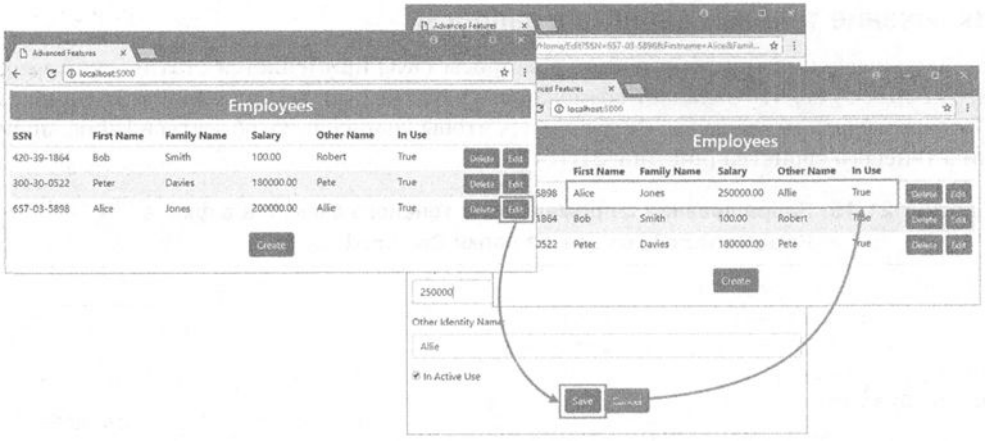

Рис. 21.7. Использование теневого свойства для упорядочения объектов в запросе LINQ

Листинг 21.16. Конфигурирование стандартного значения в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {
  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) {}

    public DbSet<Employee> Employees { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);
      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);
      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Для указания стандартного значения, которое будет применяться при создании новой строки в БД, используется метод `HasDefaultValue()`. Стандартное значение можно переопределить, предоставляя значение для свойства при сохранении объекта, но если значение не задано, то используется стандартное значение, переданное методу `HasDefaultValue()`.

Установка стандартного значения требует новой миграции. Выполните в папке проекта `AdvancedApp` команды из листинга 21.17, чтобы создать миграцию по имени `DefaultValue` и применить ее к БД.

Листинг 21.17. Создание и применение миграции к БД

```
dotnet ef migrations add DefaultValue
dotnet ef database update
```

Заглянув в метод `Up()` в файле `<отметка времени>_DefaultValue.cs`, который был создан в папке `Migrations`, вы увидите, что стандартное значение, ассоциированное со столбцом `LastUpdated`, изменилось на 1 января 2000 года, отражая указанную в листинге 21.16 дату:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.AlterColumn<DateTime>(
    name: "LastUpdated",
    table: "Employees",
    nullable: false,
    defaultValue: new DateTime(2000, 1, 1, 0, 0, 0, 0,
                               DateTimeKind.Unspecified),
    oldClrType: typeof(DateTime));
}
...
```

Отображение стандартного значения

Чтобы увязать текущий пример с остальными частями приложения, можно отобразить значение свойства `LastUpdate` для пользователя. Повысьте свойство `LastUpdated` с теневого до такого, к которому могут иметь доступ остальные части приложения, модифицировав класс `Employee`, как показано в листинге 21.18. Такая модификация не изменяет поведение свойства, а лишь означает, что к значению свойства не придется обращаться через объект контекста.

Листинг 21.18. Добавление свойства в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary;
```

```
      set {
        if (value % 2 == 0) {
          databaseSalary = value;
        }
      }
    }

    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;

    public DateTime LastUpdated { get; set; }
  }
}
```

Измените метод действия `Update()` контроллера `Home`, чтобы закомментировать оператор, который устанавливает свойство `LastUpdated` при сохранении новых объектов, и взамен полагаться на стандартное значение (листинг 21.19).

Листинг 21.19. Отключение оператора в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Update(Employee employee, decimal salary) {
  Employee existing = context.Employees.Find(employee.SSN,
    employee.FirstName, employee.FamilyName);
  if (existing == null) {
    // context.Entry(employee)
    //   .Property("LastUpdated").CurrentValue = System.DateTime.Now;
    context.Add(employee);
  } else {
    existing.Salary = salary;
    context.Entry(existing)
      .Property("LastUpdated").CurrentValue = System.DateTime.Now;
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Наконец, добавьте колонку к представлению `Index`, используемому контроллером `Home`, для отображения значения свойства `LastUpdated` (листинг 21.20).

Листинг 21.20. Добавление колонки в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
```

```
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th>In Use</th>
      <th>Last Updated</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td>@(e.OtherIdentity?.InActiveUse.ToString() ?? "(N/A)")</td>
        <td>@e.LastUpdated.ToLocalTime()</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <button type="submit" asp-action="Delete" formmethod="post"
                    class="btn btn-sm btn-danger">Delete</button>
            <button type="submit" asp-action="Edit" formmethod="get"
                    class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Чтобы увидеть стандартное значение, запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Create. Заполните поля формы и щелкните на кнопке Save; вы заметите, что свойству `LastUpdated` нового объекта присвоено значение даты 1 января 2000 года — стандартное значение, использованное в листинге 21.16 (рис. 21.8). Отображаемые даты будут относиться к стандартной локали системы; даты на рис. 21.8 имеют формат, принятый в Соединенном Королевстве.

Рис. 21.8. Присваивание стандартного значения

Совет. Если БД содержала объекты `Employee` до добавления столбца `LastUpdated` в таблицу `Employee` и они не редактировались, тогда вы увидите в `LastUpdated` значение 1 января 0001 года, как показано на рис. 21.8. Причина в том, что стандартное значение применяется только при создании новых объектов. При добавлении столбца `LastUpdated` данным, которые уже существовали в БД, было присвоено значение, состоящее из всех нулей, и именно оно отобразилось.

Обнаружение параллельных обновлений

Когда пользователь редактирует данные, большинство приложений ASP.NET Core MVC следуют циклу "запрос и обновление". Существующие значения данных извлекаются из БД в запросе для снабжения пользователя начальным состоянием, и затем на основе внесенных изменений делается обновление. Когда несколько пользователей выполняют такой цикл параллельно, может возникнуть проблема, как иллюстрируется на рис. 21.9.

Клиент А и клиент Б запрашивают те же самые данные и получают те же самые значения. Клиент А выполняет обновление и вскоре после него это делает клиент Б.

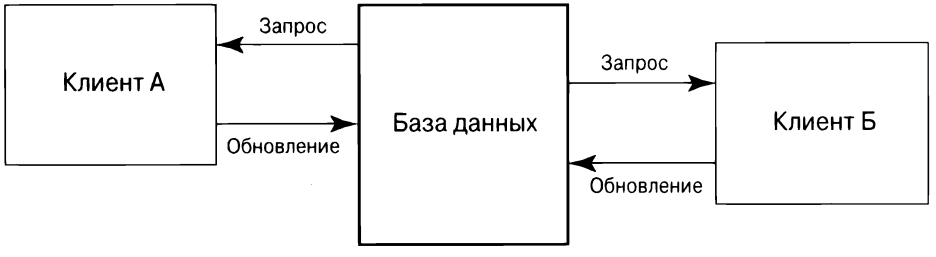

Рис. 21.9. Параллельные обновления

Такие действия могут вызвать целый диапазон проблем, включая обновления, которые тихо переписываются, и исключения для внешне допустимых обновлений. Отдельные проблемы могут проявиться спустя некоторое время по мере того, как клиенты продолжают работу с несогласованными или неполными данными.

Использование маркеров параллелизма

Инфраструктуру Entity Framework Core можно настроить на проверку значения, ассоциированного с объектом, чтобы удостовериться в отсутствии его изменения с момента чтения данных. Выбранное для такой проверки свойство называется *маркером параллелизма*, и этот прием полезен, когда вносить изменения в БД нежелательно или невозможно с целью предотвращения параллельных обновлений. Сконфигурируйте модель данных так, чтобы сделать свойство `Salary` маркером параллелизма для обновлений объектов `Employee`, как показано в листинге 21.21.

Внимание! Как вы узнаете позже, средство маркера параллелизма обладает рядом серьезных ограничений. Если у вас есть возможность модификации БД, то доступна более удачная альтернатива, которая описывается в следующем разделе.

Листинг 21.21. Создание маркера параллелизма в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field)
        .IsConcurrencyToken();

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });
```

```
      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Метод `IsConcurrencyToken()` применяется для сообщения инфраструктуре Entity Framework Core о необходимости включения существующего значения свойства в запрос при выполнении обновления, чтобы убедиться в том, что оно не изменилось. Маркеры параллелизма эффективны, только если инфраструктуре Entity Framework Core перед выполнением обновления известно старое значение, что в приложениях ASP.NET Core MVC требует небольшой работы с объектами, созданными связывателем моделей MVC.

Совет. Для сообщения инфраструктуре Entity Framework Core о необходимости использования какого-то свойства в качестве маркера параллелизма можно применять атрибут `ConcurrencyCheck`.

Оставлять пример с блоком `set`, не всегда обновляющим поддерживающее свойство, с которым он ассоциирован, нежелательно. Упростите блок `set` для свойства `Salary`, чтобы он просто обновлял поле `databaseSalary` (листинг 21.22).

Листинг 21.22. Упрощение блока `set` для свойства в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary;
      set => databaseSalary = value;
    }

    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;

    public DateTime LastUpdated { get; set; }
  }
}
```

Чтобы предоставить инфраструктуре Entity Framework Core старое значение `Salary` для проверки, добавьте скрытый элемент `input` в представление `Edit.cshtml`, используемое контроллером `Home` (листинг 21.23). Это обеспечит получение методом действия `Update()` значения свойства `Salary`, которое было прочитано из БД.

Листинг 21.23. Добавление элемента в файле `Edit.cshtml` из папки `Views/Home`

```
@model Employee
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h4 class="bg-info p-2 text-center text-white">
  Create/Edit
</h4>
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" name="originalSalary" value="@Model.Salary" />
  <div class="form-group">
    <label class="form-control-label" asp-for="SSN"></label>
    <input class="form-control" asp-for="SSN" readonly="@Model.SSN"/>
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FirstName"></label>
    <input class="form-control" asp-for="FirstName"
        readonly="@Model.FirstName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FamilyName"></label>
    <input class="form-control" asp-for="FamilyName"
         readonly="@Model.FamilyName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="Salary"></label>
    <input class="form-control" asp-for="Salary"/>
  </div>
  <input type="hidden" asp-for="OtherIdentity.Id"/>
  <div class="form-group">
    <label class="form-control-label">Other Identity Name:</label>
    <input class="form-control" asp-for="OtherIdentity.Name"/>
  </div>
  <div class="form-check">
    <label class="form-check-label">
      <input class="form-check-input" type="checkbox"
          asp-for="OtherIdentity.InActiveUse"/>
       In Active Use
    </label>
  </div>
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Save</button>
    <a class="btn btn-secondary" asp-action="Index">Cancel</a>
  </div>
</form>
```

Чтобы предоставить инфраструктуре Entity Framework Core возможность работы с исходным значением `Salary` при выполнении проверки, связанной с параллелизмом, добавьте к методу действия `Update()` контроллера `Home` параметр, который будет по-

лучать значение из элемента input и применять его к объекту Employee, используемому для обновления (листинг 21.24).

Листинг 21.24. Обновление метода действия в файле HomeController.cs из папки Controllers

```
...
[HttpPost]
public IActionResult Update(Employee employee, decimal originalSalary) {
  if (context.Employees.Count(e => e.SSN == employee.SSN
      && e.FirstName == employee.FirstName
      && e.FamilyName == employee.FamilyName) == 0) {
    context.Add(employee);
  } else {
    Employee e = new Employee {
      SSN = employee.SSN, FirstName = employee.FirstName,
      FamilyName = employee.FamilyName, Salary = originalSalary
    };
    context.Employees.Attach(e);
    e.Salary = employee.Salary;
    e.LastUpdated = DateTime.Now;
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Важнее всего запомнить, что запрашивание БД в методе Update() свело бы на нет цель маркера параллелизма, поскольку запрос возвратит значение, которое в текущее время находится в БД, а не значение, которое было сохранено в момент, когда клиент запросил редактируемые пользователем данные.

В методе Update() создается новый объект Employee и его свойству Salary присваивается значение из скрытого элемента input. Тем самым устанавливаются исходные данные для процесса обнаружения изменений Entity Framework Core с применением значений, которые были текущими, когда пользователь начинал операцию редактирования. С помощью метода Attach() объект Employee помещается под контроль системы управления изменениями, после чего свойство Salary изменяется с использованием значения, полученного в HTTP-запросе POST.

Описанная последовательность действий неудобна, но она позволяет применять маркер параллелизма в приложении ASP.NET Core MVC и обретет больший смысл, когда станет понятно, как работает это средство.

При использовании маркера параллелизма миграция БД не требуется, т.к. он затрагивает только запросы, отправляемые инфраструктурой Entity Framework Core, а не самой БД. Запустите приложение с применением dotnet run, перейдите по ссылке http://localhost:5000, щелкните на кнопке Edit для одного из объектов, отображаемых в таблице, измените значение в поле Salary и щелкните на кнопке Save.

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то увидите SQL-команду, которую инфраструктура Entity Framework Core отправила БД для выполнения обновления:

```
...
UPDATE [Employees] SET [LastUpdated] = @p0, [Salary] = @p1
WHERE [SSN] = @p2 AND [FirstName] = @p3 AND [FamilyName] = @p4 AND
[Salary] = @p5;
...
```

Конструкция `WHERE`, выделенная полужирным, ограничивает обновление так, что оно будет применяться только к строке в таблице `Employees`, которая имеет конкретный составной ключ и определенное значение `Salary`. Это воспрепятствует применению обновления, если другой клиент модифицировал маркер параллелизма, потому что ни одна строка не будет соответствовать конструкции `WHERE` оператора `UPDATE`. Инфраструктура Entity Framework Core проверяет, сколько строк было изменено оператором `UPDATE`. Если обновилась одна строка, тогда она полагает, что параллельных обновлений не было. Если ни одной строки не обновилось, то Entity Framework Core считает, что маркер параллелизма был изменен другим клиентом, и сообщает об ошибке.

Чтобы увидеть сообщение об ошибке, откройте второе окно браузера и выполните перемежающееся обновление: щелкните на кнопке Edit для того же самого объекта `Employee`, который редактируется в первом окне браузера, измените значение в поле Salary в обоих окнах и щелкните на кнопке Save в обоих окнах. Второе обновление потерпит неудачу, и вы получите сообщение об ошибке, показанное на рис. 21.10.

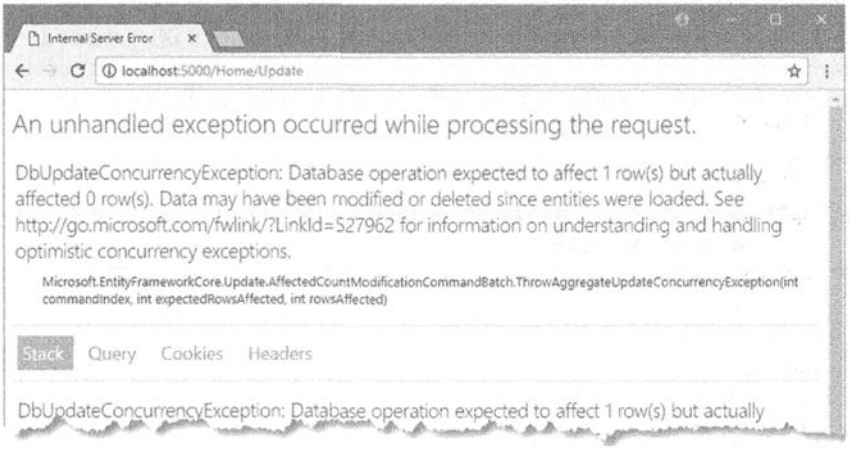

Рис. 21.10. Результат проверки параллелизма

Ограничение средства проверки параллелизма связано с тем, что каждое обновление должно модифицировать тот же самый столбец, чтобы указать на изменение, и каждое обновление обязано знать, что именно служит признаком изменения этого столбца. Таким образом, в примере приложения обновления должны модифицировать свойство `Salary`, чтобы дать сигнал другим клиентам о наличии изменения; обновления, которые воздействуют только на другие свойства, не препятствуют параллельным обновлениям. Несмотря на упомянутые проблемы, использовать маркер параллелизма может быть удобно, если у вас нет возможности модифицировать БД, но вы в состоянии гарантировать, что клиенты будут каждый раз обновлять определенное свойство.

Избегание ловушки, связанной с датой

Вы можете предположить, что ограничения маркера параллелизма удастся обойти с применением свойства `LastUpdated`, которое обновляется при каждом обновлении объекта `Employee`, и не надеяться на то, что пользователь внесет определенное изменение. К сожалению, вам придется гарантировать, что инфраструктура Entity Framework Core будет запрашивать именно то значение, которое хранится в БД, а даты подвержены вариациям в плане точности и форматирования. Значения `LastUpdated` хранятся в БД примерно так:

```
2017-11-10 09:11:42.3366667
```

Но когда Entity Framework Core читает эти значения и преобразует их в объект `DateTime`, точность теряется и формат изменяется согласно локали. В лучшем случае вы получите значение вроде следующего:

```
10/11/2017 09:11:42
```

При выполнении инфраструктурой Entity Framework Core оператора `UPDATE` значение даты, указанное в конструкции `WHERE`, не совпадет со значением в БД, а потому ни одна запись не подойдет. Инфраструктура Entity Framework Core предположит, что проверка параллелизма не прошла, и вы получите исключение, показанное на рис. 21.10.

Использование версии строки для обнаружения параллельных обновлений

Если в проекте имеется возможность внесения изменений в БД, тогда более надежной альтернативой будет применение *версии строки*, представляющей собой отметку времени, которая автоматически обновляется, когда происходит обновление, но хранится так, что отличия в форматировании не возникают. Добавьте в класс `Employee` свойство, которое будет использоваться для ведения версий строк (листинг 21.25).

Листинг 21.25. Добавление свойства для ведения версий строк в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary;
      set => databaseSalary = value;
    }
    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;
    public DateTime LastUpdated { get; set; }
    public byte[] RowVersion { get; set; }
  }
}
```

Ведение версий строк реализуется с применением свойства типа массива `byte`, которое устраняет проблемы с точностью и форматированием данных. Добавьте оператор Fluent API, который конфигурирует новое свойство, чтобы оно использовалось средством ведения версий строк (листинг 21.26). Кроме того, закомментируйте вызов метода, настраивающий свойство `Salary` в качестве маркера параллелизма.

На заметку! Сообщить инфраструктуре Entity Framework Core о свойстве версии строки можно путем применения к нему атрибута `TimeStamp`, что эквивалентно использованию метода `IsRowVersion()` из Fluent API.

Листинг 21.26. Добавление свойства версии строки в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);
        // .IsConcurrencyToken();

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Property(e => e.RowVersion).IsRowVersion();

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName });

      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Средство ведения версий строк конфигурируется путем выбора свойства и вызова метода `IsRowVersion()`. Для предотвращения параллельных обновлений значение свойства `RowVersion`, которое хранилось в БД в начале операции редактирования, должно быть включено в HTML-форму, отправляемую клиенту, чтобы его мог получить метод `Update()`, когда пользователь отправляет изменение, как делалось в примере из предыдущего раздела. Добавьте в представление `Edit.cshtml` скрытый элемент `input`, который содержит значение свойства `RowVersion` (листинг 21.27).

Листинг 21.27. Добавление элемента в файле `Edit.cshtml` из папки `Views/Home`

```
@model Employee
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h4 class="bg-info p-2 text-center text-white">
  Create/Edit
</h4>
<form asp-action="Update" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="RowVersion" />
  @*<input type="hidden" name="originalSalary" value="@Model.Salary"
/>*@
  <div class="form-group">
    <label class="form-control-label" asp-for="SSN"></label>
    <input class="form-control" asp-for="SSN" readonly="@Model.SSN" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FirstName"></label>
    <input class="form-control" asp-for="FirstName"
          readonly="@Model.FirstName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="FamilyName"></label>
    <input class="form-control" asp-for="FamilyName"
          readonly="@Model.FamilyName" />
  </div>
  <div class="form-group">
    <label class="form-control-label" asp-for="Salary"></label>
    <input class="form-control" asp-for="Salary" />
  </div>
  <input type="hidden" asp-for="OtherIdentity.Id" />
  <div class="form-group">
    <label class="form-control-label">Other Identity Name:</label>
    <input class="form-control" asp-for="OtherIdentity.Name" />
  </div>
  <div class="form-check">
    <label class="form-check-label">
    <input class="form-check-input" type="checkbox"
          asp-for="OtherIdentity.InActiveUse" />
      In Active Use
    </label>
  </div>
```

```
  <div class="text-center">
    <button type="submit" class="btn btn-primary">Save</button>
    <a class="btn btn-secondary" asp-action="Index">Cancel</a>
  </div>
</form>
```

Для использования значения, включенного в HTTP-запрос POST, модифицируйте метод действия Update() контроллера Home, как показано в листинге 21.28.

Обновление выполняется с помощью такого же приема, как в предыдущем примере, и важно не запрашивать у БД текущее значение RowVersion, что разрушит сам замысел проверки параллелизма.

Листинг 21.28. Применение версии строки в файле HomeController.cs из папки Controllers

```
...
[HttpPost]
public IActionResult Update(Employee employee) {
  if (context.Employees.Count(e => e.SSN == employee.SSN
      && e.FirstName == employee.FirstName
      && e.FamilyName == employee.FamilyName) == 0) {
    context.Add(employee);
  } else {
    Employee e = new Employee {
      SSN = employee.SSN, FirstName = employee.FirstName,
      FamilyName = employee.FamilyName, RowVersion = employee.RowVersion
    };
    context.Employees.Attach(e);
    e.Salary = employee.Salary;
    e.LastUpdated = DateTime.Now;
  }
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Обратите внимание, что при выполнении обновления изменять свойство RowVersion не нужно. Сервер баз данных будет генерировать для каждого обновления новое значение RowVersion, которое инфраструктура Entity Framework Core задействует в конструкции WHERE оператора UPDATE.

Наконец, для корректной работы функции мягкого удаления добавьте в представление Index, используемое контроллером Home, скрытый элемент input (листинг 21.29).

Совет. Обратите внимание, что для установки значения элемента input применяется вспомогательная функция дескриптора asp-for. Типом свойства RowVersion является массив byte, и указанная вспомогательная функция дескриптора сцепляет элементы массива с целью формирования строки, которую связыватель моделей MVC может разобрать в последующем HTTP-запросе POST.

Листинг 21.29. Добавление элемента в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th>In Use</th>
      <th>Last Updated</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data
    </td></tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td>@(e.OtherIdentity?.InActiveUse.ToString() ?? "(N/A)")</td>
        <td>@e.LastUpdated.ToLocalTime()</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                  asp-for="@e.RowVersion" />
            <button type="submit" asp-action="Delete" formmethod="post"
                   class="btn btn-sm btn-danger">
              Delete
            </button>
            <button type="submit" asp-action="Edit" formmethod="get"
                  class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
```

```
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Недостаток подхода с ведением версий строк связан с тем, что он требует внесения изменений в БД. Выполните в папке проекта AdvancedApp команды из листинга 21.30, чтобы создать и применить к БД миграцию по имени RowVersion.

Листинг 21.30. Создание и применение миграции к БД

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

Воспользовавшись двумя окнами браузера для выполнения перемежающихся обновлений, вы получите то же самое сообщение об ошибке, которое приводилось на рис. 21.10. Разница здесь в том, что все обновления будут инициировать изменение значения RowVersion. Это значит, что независимо от модифицируемых свойств все обновления будут причиной изменения, которое можно задействовать для обнаружения параллельных обновлений.

Резюме

В главе были описаны возможности, которые инфраструктура Entity Framework Core предлагает для взятия под контроль способа создания или сохранения данных в БД. Вы узнали, как изменять тип SQL, используемый для хранения свойства в БД, каким образом проверять достоверность или форматировать данные с применением поддерживающих полей, как использовать теневые свойства для данных, доступ к которым из других частей приложения нежелателен, и каким образом устанавливать стандартные значения, когда объекты сохраняются в БД. В заключение было показано, как обнаруживать параллельные обновления с применением маркера параллелизма и ведения версий строк. В следующей главе будут рассматриваться расширенные возможности, доступные для удаления данных.

ГЛАВА 22

Удаление данных

Удаление данных может оказаться на удивления сложной задачей, особенно когда приходится иметь дело со связанными данными или моделированием существующей БД. В этой главе рассматриваются средства Entity Framework Core для работы с удалением данных, демонстрируется функционирование каждого из них и объясняется, когда они полезны. В табл. 22.1 приведены сведения, позволяющие поместить средства для удаления данных в контекст.

Таблица 22.1. Помещение средств для удаления данных в контекст

Вопрос	Ответ
Что это такое?	Средства для удаления данных позволяют указывать, каким образом сервер баз данных реагирует на запросы удаления данных
Чем они полезны?	При удалении объекта существует несколько способов для обработки связанных данных, и выбор правильного из них будет гарантировать, что приложение имеет доступ к данным, в которых оно нуждается
Как они используются?	Средства для удаления данных применяются через комбинацию операторов Fluent API изменений классов хранилища в проекте
Существуют ли какие-то скрытые ловушки или ограничения?	Необходимо соблюдать осторожность, чтобы не вызвать в БД непредвиденное каскадное удаление и удалить больше данных, чем было запланировано. В равной степени возможно появление висячих данных из-за того, что не был удален достаточный объем данных
Существуют ли альтернативы?	Можно положиться на стандартные линии поведения, которые инфраструктура Entity Framework Core применяет к БД

В табл. 22.2 приведена сводка по главе.

Таблица 22.2. Сводка по главе

Задача	Решение	Листинг
Изменение поведения удаления	Используйте метод `OnDelete()`	22.1–22.19

Подготовительные шаги

В главе продолжается работа с проектом AdvancedApp, но в качестве подготовительных шагов потребуется внести ряд изменений. Для отображения деталей объектов SecondaryIdentity создайте в папке Views/Home файл частичного представления по имени SecondaryIdentities.cshtml с содержимым, показанным в листинге 22.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc.

Листинг 22.1. Содержимое файла SecondaryIdentities.cshtml из папки Views/Home

```
<h3 class="bg-info p-2 text-center text-white">Secondary Identities</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>Key</th>
      <th>Name</th>
      <th>Active Use</th>
      <th>Foreign SSN</th>
      <th>Foreign FamilyName</th>
      <th>Foreign FirstName</th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="6" class="text-center">No Data</td>
    </tr>
    @foreach (SecondaryIdentity ident in ViewBag.Secondaries) {
      <tr>
        <td>@ident.Id</td>
        <td>@ident.Name</td>
        <td>@ident.InActiveUse</td>
        <td>@(ident.PrimarySSN ?? "(null)")</td>
        <td>@(ident.PrimaryFirstName ?? "(null)")</td>
        <td>@(ident.PrimaryFamilyName ?? "(null)")</td>
      </tr>
    }
  </tbody>
</table>
```

Чтобы отобразить новое частичное представление, добавьте в представление Index, применяемое контроллером Home, выделенный полужирным элемент в листинге 22.2. Кроме того, удалите столбцы In Use (В употреблении) и Last Updated (Последний раз обновлялся) из таблицы, которая отображает объекты Employees.

Листинг 22.2. Добавление частичного представления в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      @*<th>In Use</th>*@
      @*<th>Last Updated</th>*@
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        @*<td>@(e.OtherIdentity?.InActiveUse.ToString() ?? "(N/A)")</td>*@
        @*<td>@e.LastUpdated.ToLocalTime()</td>*@
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                asp-for="@e.RowVersion" />
            <button type="submit" asp-action="Delete" formmethod="post"
                  class="btn btn-sm btn-danger">
              Delete
            </button>
            <button type="submit" asp-action="Edit" formmethod="get"
                  class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
```

```
@Html.Partial("SecondaryIdentities")
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Модифицируйте метод действия `Index()` контроллера `Home`, для предоставления доступа к объектам `SecondaryIdentity` через `ViewBag` (листинг 22.3). Также измените метод действия `Delete()`, чтобы заменить средство мягкого удаления обычным удалением, подобным тому, что использовалось в предшествующих главах. (В конце главы средство мягкого удаления будет восстановлено.)

Листинг 22.3. Внесение изменений в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System;
namespace AdvancedApp.Controllers {
  public class HomeController : Controller {
    private AdvancedContext context;
    public HomeController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      ViewBag.Secondaries = context.Set<SecondaryIdentity>();
      return View(context.Employees.Include(e => e.OtherIdentity)
        .OrderByDescending(e => EF.Property<DateTime>(e, "LastUpdated")));
    }
    public IActionResult Edit(string SSN, string firstName, string familyName)
    {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
      } else {
        Employee e = new Employee {
          SSN = employee.SSN, FirstName = employee.FirstName,
          FamilyName = employee.FamilyName,
          RowVersion = employee.RowVersion
        };
        context.Employees.Attach(e);
        e.Salary = employee.Salary;
        e.LastUpdated = DateTime.Now;
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
```

```
    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Remove(employee);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Выполните в папке проекта AdvancedApp команды из листинга 22.4, чтобы удалить и воссоздать БД.

Листинг 22.4. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением dotnet run, перейдите по ссылке http://localhost:5000, щелкните на кнопке Create (Создать) и сохраните в БД четыре объекта Employee, используя данные из табл. 22.3.

Таблица 22.3. Значения данных для создания объектов в примере

SSN (Номер карточки социального страхования)	First Name (Имя)	Family Name (Фамилия)	Salary (Оклад)	Other Name (Другое имя)	In Active Use (Активно используется)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
657-03-5898	Alice	Jones	200000	Allie	Флажок отмечен
300-30-0522	Peter	Davies	180000	Pete	Флажок отмечен
751-07-9405	Theodora	Francis	140000	Dora	Флажок отмечен

Результат создания четырех объектов Employee показан на рис. 22.1.

Понятие ограничений удаления

Метод Remove() контекста применяется для удаления данных из БД. Передача объекта методу Remove() сообщает инфраструктуре Entity Framework Core о том, что соответствующие данные в БД больше не нужны и при вызове метода SaveChanges() серверу баз данных должна быть отправлена команда DELETE.

Распространенная проблема в приложениях ASP.NET Core MVC возникает, когда подлежащий удалению объект имеет отношение с другими данными, хранящимися в БД. Вы можете увидеть пример, щелкнув на кнопке Delete (Удалить) для одного из объектов, которые отображает приложение. Сервер баз данных не позволит удалить данные и сгенерирует исключение, как видно на рис. 22.2.

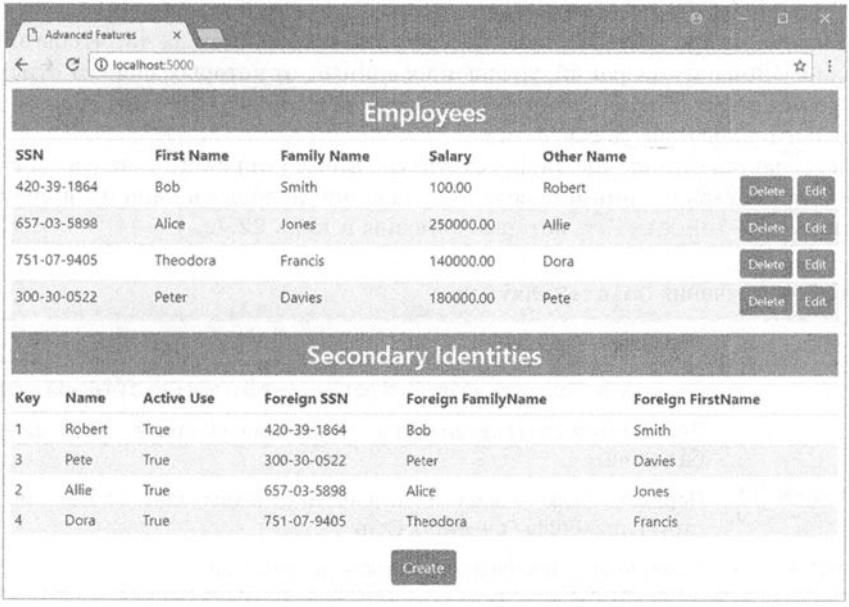

Рис. 22.1. Выполнение примера приложения

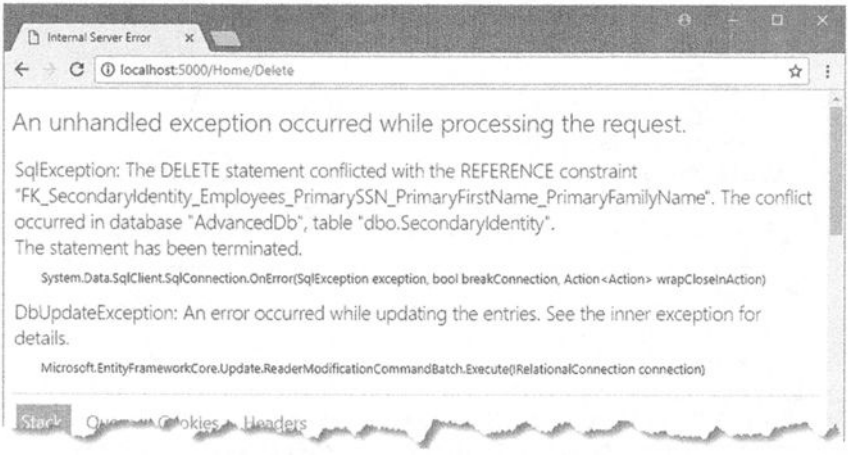

Рис. 22.2. Попытка удаления данных

При удалении главной сущности в отношении зависимая сущность попадет в одну из трех ситуаций, что основано на конфигурации модели данных.

- Зависимая сущность будет удалена. Это означает, что удаление объекта `Employee` приведет также к удалению связанного с ним объекта `SecondaryIdentity`.
- Свойства внешних ключей зависимой сущности устанавливаются в `null`. Это означает, что удаление объекта `Employee` приведет к установке в `null` свойств `SSN`, `PrimaryFirstName` и `PrimaryFamilyName` связанного объекта.
- Никаких изменений в зависимую сущность не вносится.

Третий вариант в действительности означает, что вам придется управлять связанными данными в приложении, взяв на себя ответственность за то, чтобы не предпринималось никаких операций, удаляющих данные, от которых зависит отношение. Серверы баз данных обеспечивают целостность БД и сообщают об ошибках, не разрешая создавать проблемы со ссылками.

Когда создается отношение, инфраструктура Entity Framework Core следует набору соглашений для выбора линии поведения удаления, используя одно из значений перечисления `DeleteBehavior`, которые описаны в табл. 22.4.

Таблица 22.4. Значения `DeleteBehavior`

Имя	Описание
`Cascade`	Зависимая сущность удаляется автоматически вместе с главной сущностью
`SetNull`	Первичный ключ главной сущности устанавливается в `null` сервером баз данных
`ClientSetNull`	Первичный ключ зависимой сущности устанавливается в `null` инфраструктурой Entity Framework Core
`Restrict`	Изменения в зависимую сущность не вносятся

Отношение между классами `Employee` и `SecondaryIdentity` конфигурируется в миграции `CompositeKey`. Заглянув в метод `Up()` внутри файла `<отметка времени>_CompositeKey.cs` в папке `Migrations`, вы обнаружите оператор, который устанавливает поведение удаления:

```
...
migrationBuilder.CreateTable(name: "SecondaryIdentity",
  columns: table => new {
    Id = table.Column<long>(type: "bigint", nullable: false)
      .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn),
    ...для краткости определения столбцов не показаны...
  },
  constraints: table => {
    table.PrimaryKey("PK_SecondaryIdentity", x => x.Id);
    table.ForeignKey(name: "FK_SecondaryIdentity_Employees_
      PrimarySSN_PrimaryFirstName_PrimaryFamilyName",
        columns: x => new { x.PrimarySSN, x.PrimaryFirstName,
          x.PrimaryFamilyName },
        principalTable: "Employees",
        principalColumns: new[] { "SSN", "FirstName", "FamilyName" },
        onDelete: ReferentialAction.Restrict);
  });
...
```

Отношение сконфигурировано с поведением `Restrict`.

В объект `SecondaryIdentity` не вносится никаких изменений при удалении его связанного объекта `Employee`, что вызывает исключение, показанное на рис. 22.2.

Конфигурирование поведения удаления

Самый надежный способ получить требуемое поведение предусматривает выбор желаемого поведения удаления с применением Fluent API, явно конфигурируя модель данных со значением `DeleteBehavior`. Описания в табл. 22.4 могут казаться достаточно простыми, но существуют некоторые сложности. Эти линии поведения могут отличаться для обязательных и необязательных отношений. Вдобавок, просто чтобы усложнить задачу, инфраструктура Entity Framework Core будет выполнять лишь некоторые операции на связанных данных, загруженных из БД, что может стать источником путаницы в приложении ASP.NET Core MVC, где объекты часто создаются связывателем моделей MVC и требуют специальной обработки. В последующих разделах будет объяснена работа каждой линии поведения удаления.

Принятие решения о том, какую линию поведения удаления использовать

Выбор линии поведения удаления способен сбивать с толку, а определить, какая из них нужна, может быть нелегко даже после ознакомления с демонстрациями каждой линии поведения в последующих разделах. Если вы не уверены, с чего начинать, то вот вам мой совет.

Если вы имеете дело с обязательным отношением, тогда должны начинать с поведения `Cascade`, но провести некоторое тестирование, чтобы удостовериться в том, что одиночная операция удаления не распространяется по всей БД и не удаляет больше данных, чем ожидалось.

Если вы работаете с необязательным отношением, то должны начинать с поведения `SetNull`, когда сервер баз данных его поддерживает, и поведения `ClientSetNull` в противном случае. Но если вы не планируете иметь дело с висячими данными, тогда взамен применяйте поведение `Cascade`.

Избегайте поведения `Restrict`, если только вы не моделируете БД, имеющую специальные требования, которые не могут быть решены с использованием других линий поведения. Поведение `Restrict` позволяет получить полный контроль над процессом удаления, но его реализация вопреки ожиданиям может оказаться труднее и вдобавок в процессе легко допустить ошибки.

Использование каскадного удаления

Значение `DeleteBehavior.Cascade` конфигурирует БД так, чтобы зависимые сущности удалялись из БД, когда удаляется главная сущность, с которой они связаны. Работать с каскадным удалением проще всего, но требуется соблюдать осторожность, поскольку можно легко удалить больше данных, чем ожидалось, т.к. сервер баз данных продолжит следовать по отношениям и удалять данные для обеспечения целостности БД. Если вы применяете каскадное удаление слишком свободно, то можете обнаружить, что операция удаления способна выходить из-под контроля и приводить к далеко идущим последствиям.

Добавьте в класс контекста оператор Fluent API, выбирающий поведение каскадного удаления для отношения `Employee/SecondaryIdentity` (листинг 22.5).

На заметку! Поведение удаления для отношения может быть указано только с использованием Fluent API. Поддержка со стороны атрибутов для выбора поведения удаления отсутствует.

Листинг 22.5. Конфигурирование поведения удаления в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);
        // .IsConcurrencyToken();

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Property(e => e.RowVersion).IsRowVersion();

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName })
        .OnDelete(DeleteBehavior.Cascade);

      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Поведение удаления для отношения конфигурируется с применением метода `OnDelete()`, который принимает значение `DeleteBehavior` в качестве своего аргумента. Этот метод используется как часть цепочки обращений к методам, определяющих отношение между двумя классами.

Выполните в папке проекта `AdvancedApp` команды из листинга 22.6 для создания и применения к БД миграции, которая изменит поведение удаления.

Листинг 22.6. Создание и применение миграции к БД

```
dotnet ef migrations add CascadeDelete
dotnet ef database update
```

Чтобы увидеть эффект от изменения, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete для одного из объектов `Employee`. Объект `Employee` и связанный с ним объект `SecondaryIdentity` будут удалены из БД (рис. 22.3).

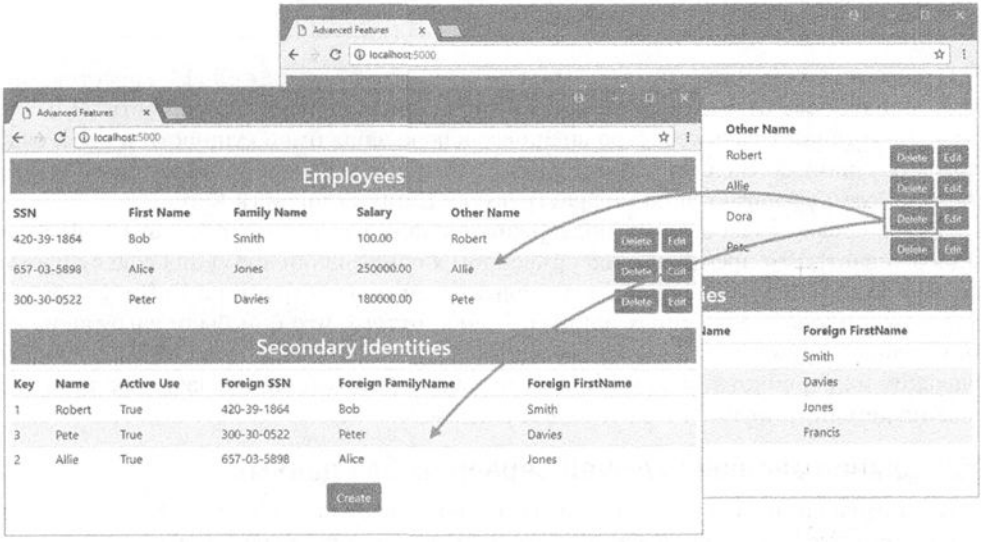

Рис. 22.3. Применение каскадного удаления

Совет. Если у вас закончатся объекты для удаления, тогда щелкните на кнопке Create, заполните поля HTML-формы и щелкните на кнопке Save (Сохранить), чтобы сохранить новые данные в БД. Выбранные вами значения роли не играют, т.к. внимание во всех примерах, рассматриваемых в главе, сосредоточено на удалении данных.

Каскадное удаление реализуется сервером баз данных, который автоматически удаляет связанные данные из БД. Вы можете удостовериться в этом, просмотрев журнальные сообщения, которые сгенерировало приложение. В результате щелчка на кнопке Delete инфраструктура Entity Framework Core отправляет серверу БД следующий SQL-запрос:

```
...
DELETE FROM [Employees]
WHERE [SSN] = @p0 AND [FirstName] = @p1 AND [FamilyName] = @p2
AND [RowVersion] = @p3;
...
```

Инфраструктура Entity Framework Core удаляет только главную сущность и может полагаться на то, что сервер баз данных удалит связанные данных с целью предохранения целостности БД.

Установка внешних ключей в null

Два значения, SetNull и ClientSetNull, в перечислении DeleteBehavior реагируют на удаление главной сущности тем, что оставляют зависимую сущность в БД, но устанавливают внешний ключ в null. Результатом является разрыв отношения между двумя объектами, так что главная сущность может быть удалена без нарушения ссылочной целостности БД.

Внимание! Такое поведение может использоваться только для необязательных отношений. Применение этого поведения для обязательного отношения приведет к генерации исключения.

Отличие между линиями поведения SetNull и ClientSetNull касается того, кто берет на себя ответственность за модификацию зависимой сущности. В случае SetNull устанавливать свойство внешнего ключа зависимой сущности в null будет сервер баз данных. В случае ClientSetNull ответственность за обновление зависимой сущности возлагается на инфраструктуру Entity Framework Core.

Не все серверы баз данных поддерживают поведение SetNull (хотя SQL Server поддерживает). Его преимущество объясняется согласованностью, поскольку поведение ClientSetNull будет работать, только если инфраструктура Entity Framework Core осведомлена о связанных данных — либо оттого, что они были загружены как часть запроса, либо потому, что связыватель моделей MVC создал объект, который содержит их первичный ключ. Разные способы использования указанных линий поведения демонстрируются в следующем разделе.

Обновление внешних ключей сервером баз данных

Если имеющийся сервер баз данных поддерживает поведение SetNull, тогда его можно применять для обновления зависимых сущностей, даже при условии, что они не загружаются инфраструктурой Entity Framework Core. Из двух линий поведения, устанавливающих внешние ключи в null, поведение SetNull проще в работе, т.к. не приходится удостоверяться в том, что Entity Framework Core отслеживает объекты, которые должны модифицироваться. Выберите поведение SetNull для отношения Employee/SecondaryIdentity, изменив аргумент метода OnDelete(), как показано в листинге 22.7.

Листинг 22.7. Изменение поведения удаления в файле AdvancedContext.cs из папки Models

```
...
modelBuilder.Entity<SecondaryIdentity>()
  .HasOne(s => s.PrimaryIdentity)
  .WithOne(e => e.OtherIdentity)
  .HasPrincipalKey<Employee>(e => new { e.SSN,
    e.FirstName, e.FamilyName })
  .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
    s.PrimaryFirstName, s.PrimaryFamilyName })
  .OnDelete(DeleteBehavior.SetNull);
...
```

Выполните в папке проекта AdvancedApp команды из листинга 22.8 для создания новой миграции и ее применения к БД.

Листинг 22.8. Создание и применение миграции к БД

```
dotnet ef migrations add SetNullDelete
dotnet ef database update
```

Чтобы посмотреть, как работает поведение удаления, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на одной из кнопок Delete. Вы увидите, что объект `Employee` удалится из БД и свойства внешних ключей связанного объекта `SecondaryIdentity` установятся в `null` (рис. 22.4).

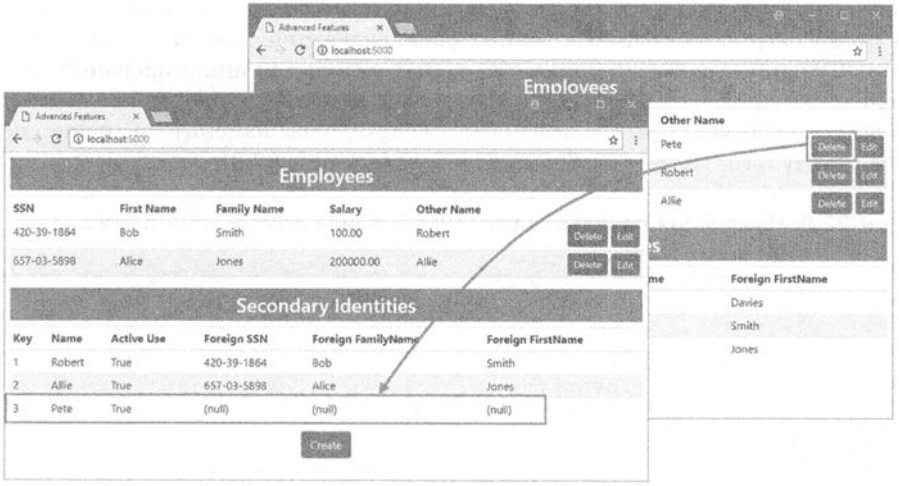

Рис. 22.4. Применение поведения `SetNull`

Если вы заглянете в журнальные сообщения, сгенерированные приложением, то заметите команду `DELETE`, которую инфраструктура Entity Framework Core использовала для удаления объекта `Employee`:

```
...
DELETE FROM [Employees]
WHERE [SSN] = @p0 AND [FirstName] = @p1 AND [FamilyName] = @p2
  AND [RowVersion] = @p3;
...
```

Никаких команд для обновления связанного объекта `SecondaryIdentity` не отправлялось, потому что за установку в `null` свойств внешних ключей любых зависимых сущностей несет ответственность сервер баз данных.

Внимание! Поскольку поведение `SetNull` не предусматривает удаление зависимых сущностей из БД, могут появиться "висячие" данные. Вы должны применять такое поведение, только если приложение, вероятно, будет повторно использовать зависимые сущности, оставшиеся в БД. Что касается примера приложения, то в БД теперь присутствует объект `SecondaryIdentity`, который не ассоциирован с каким-либо объектом `Employee`, а приложение не предлагает никаких средств для указания существующего объекта при создании нового объекта `Employee`, фактически приводя к появлению висячих данных. Висячие данные могут вызывать неожиданные проблемы, особенно в случае применения ограничений уникальности к естественным ключам, которые воспрепятствуют созданию новых объектов с такими же ключами, как у висячих объектов.

Обновление внешних ключей инфраструктурой Entity Framework Core

В случае выбора поведения `ClientSetNull` обновлять свойства внешних ключей зависимых сущностей при удалении главной сущности будет инфраструктура Entity Framework Core. Поступать так полезно, когда имеющийся сервер баз данных не поддерживает поведение `SetNull`, хотя в приложениях ASP.NET Core MVC потребуется выполнить дополнительную работу, потому что Entity Framework Core будет обновлять только те объекты, которые загружались из БД или добавлялись под контроль отслеживания изменений вручную. Это означает либо использование добавочного запроса для загрузки связанных данных, либо включение дополнительной информации в HTTP-запрос, который пользователь отправляет для инициирования операции удаления.

Чтобы сконфигурировать поведение `ClientSetNull`, измените значение, передаваемое методу `OnDelete()` в классе контекста (листинг 22.9).

Листинг 22.9. Изменение поведения удаления в файле `AdvancedContext.cs` из папки `Models`

```
...
modelBuilder.Entity<SecondaryIdentity>()
  .HasOne(s => s.PrimaryIdentity)
  .WithOne(e => e.OtherIdentity)
  .HasPrincipalKey<Employee>(e => new { e.SSN,
    e.FirstName, e.FamilyName })
  .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
    s.PrimaryFirstName, s.PrimaryFamilyName })
  .OnDelete(DeleteBehavior.ClientSetNull);
...
```

Для конфигурирования БД нужна новая миграция. Выполните в папке проекта `AdvancedApp` команды из листинга 22.10 для создания и применения миграции к БД.

Листинг 22.10. Создание и применение миграции к БД

```
dotnet ef migrations add ClientSetNullDelete
dotnet ef database update
```

Если вы просмотрите метод `Up()` в файле `<отметка времени>_ClientSetNull Delete.cs`, добавленном в папку `Migrations`, то увидите, что поведение удаления изменилось:

```
...
protected override void Up(MigrationBuilder migrationBuilder) {
  migrationBuilder.DropForeignKey(
    name: "FK_SecondaryIdentity_Employees
      _PrimarySSN_PrimaryFirstName_PrimaryFamilyName",
    table: "SecondaryIdentity");

  migrationBuilder.AddForeignKey(
    name: "FK_SecondaryIdentity_Employees
      _PrimarySSN_PrimaryFirstName_PrimaryFamilyName",
```

```
            table: "SecondaryIdentity",
            columns: new[] { "PrimarySSN", "PrimaryFirstName", "PrimaryFamilyName" },
            principalTable: "Employees",
            principalColumns: new[] { "SSN", "FirstName", "FamilyName" },
            onDelete: ReferentialAction.Restrict);
    }
    ...
```

Миграция изменяет поведение обратно на `Restrict`, так что сервер баз данных не предпринимает никаких действий, когда объект `Employee` удаляется. Это имеет смысл, поскольку поведение `ClientSetNull` полагается на то, что значения внешних ключей обновит инфраструктура Entity Framework Core, а не сервер баз данных.

Запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete; вы получите сообщение об ошибке, показанное на рис. 22.5.

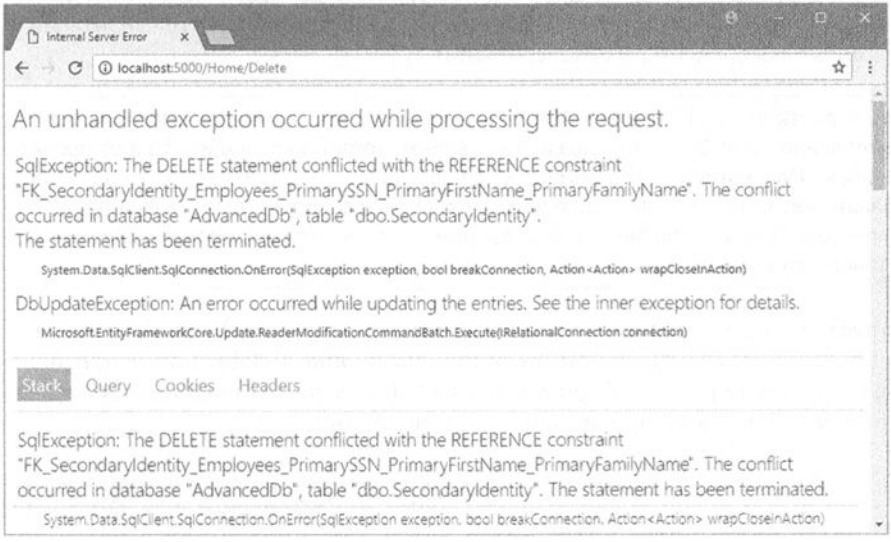

Рис. 22.5. Удаление главной сущности

Так происходит, если выбрать поведение `ClientSetNull`, но не предоставить инфраструктуре Entity Framework Core доступ к зависимым сущностям, которые связаны с удаляемым объектом. Эффект эквивалентен указанию поведения `Restrict`, потому что сервер баз данных не вносит никаких изменений в зависимые сущности, а инфраструктуре Entity Framework Core ничего о них не известно.

Использование запроса для загрузки связанных данных

Один из способов сообщения инфраструктуре Entity Framework Core о наличии связанных объектов, подлежащих удалению, предусматривает выполнение запроса к БД для загрузки данных (листинге 22.11).

Листинг 22.11. Запрашивание связанных данных в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Delete(Employee employee) {
  context.Set<SecondaryIdentity>().FirstOrDefault(id =>
    id.PrimarySSN == employee.SSN
    && id.PrimaryFirstName == employee.FirstName
    && id.PrimaryFamilyName == employee.FamilyName);
  context.Employees.Remove(employee);
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Такого запроса достаточно для загрузки данных, которые будут удалены, и с объектами, создаваемыми инфраструктурой Entity Framework Core, делать ничего не нужно кроме обеспечения их отслеживания.

Совет. В листинге 22.11 применялся метод `FirstOrDefault()`, выполняющий запрос немедленно. Если доступ к связанным данным можно производить только посредством запроса LINQ, который создает объект реализации `IQueryable<T>`, тогда потребуется вызвать метод `Load()` для принудительной оценки запроса. Иначе инфраструктура Entity Framework Core не выполнит запрос, данные не загрузятся и свойства внешних ключей не установятся в `null`.

Запустите приложение с использованием `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete. Как и при поведении `SetNull` объект `Employee` будет удален из БД, а свойства внешних ключей объекта `SecondaryIdentity` установятся в `null` (рис. 22.6).

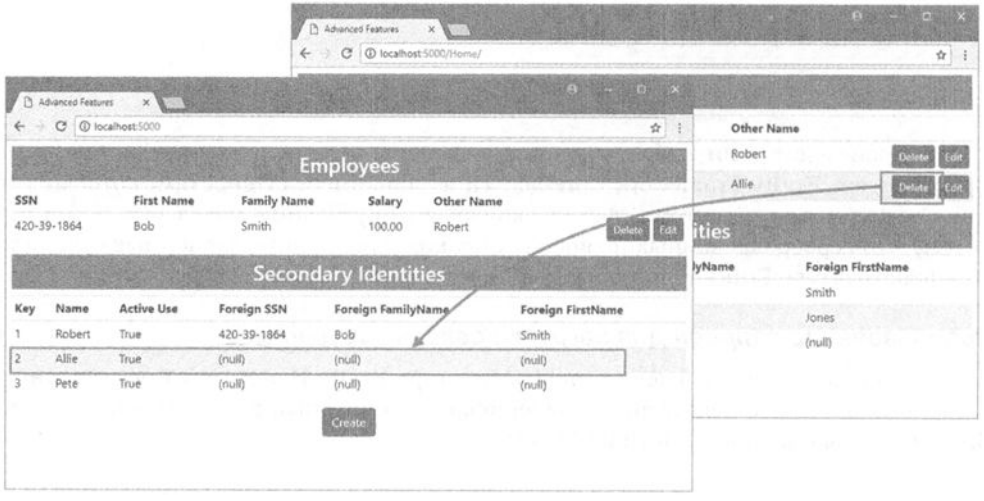

Рис. 22.6. Применение поведения `ClientSetNull`

Если вы просмотрите журнальные сообщения, сгенерированные приложением, то увидите, что инфраструктура Entity Framework Core взяла на себя ответственность за работу с обоими объектами. Первая команда устанавливает значения внешних ключей объекта SecondaryIdentity в null:

```
...
UPDATE [SecondaryIdentity] SET [PrimaryFamilyName] = @p0,
[PrimaryFirstName] = @p1,
  [PrimarySSN] = @p2
WHERE [Id] = @p3;
...
```

Команда гарантирует, что объект Employee может быть удален, не вызывая проблем со ссылочной целостностью. Операция удаления выполняется с помощью второй команды:

```
...
DELETE FROM [Employees]
WHERE [SSN] = @p4 AND [FirstName] = @p5 AND [FamilyName] = @p6
  AND [RowVersion] = @p7;
...
```

Избегание запроса связанных данных

Для выполнения обновления инфраструктуре Entity Framework Core необходим только первичный ключ зависимой сущности. Избежать добавочного запроса в листинге 22.11 можно путем включения дополнительного значения в HTTP-запрос POST, который нацелен на действие Delete, и использовать его для снабжения Entity Framework Core требуемой информацией. Добавьте в представление Index элемент для первичного ключа связанного объекта SecondaryIdentity (листинг 22.12).

Листинг 22.12. Добавление элемента в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
```

```
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                asp-for="@e.RowVersion" />
            <input type="hidden" name="OtherIdentity.Id"
                value="@e.OtherIdentity.Id" />
          <button type="submit" asp-action="Delete" formmethod="post"
                class="btn btn-sm btn-danger">
            Delete
          </button>
          <button type="submit" asp-action="Edit" formmethod="get"
                class="btn btn-sm btn-primary">
            Edit
          </button>
        </form>
      </td>
    </tr>
  }
 </tbody>
</table>
@Html.Partial("SecondaryIdentities")
<div class="text-center">
 <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Закомментируйте запрос в методе действия `Delete()`, как иллюстрируется в листинге 22.13. Новое значение, включенное в HTTP-запрос `POST`, позволяет связывателю моделей MVC создать объект `SecondaryIdentity`, который инфраструктура Entity Framework Core будет применять для обновления БД перед выполнением операции удаления.

Листинг 22.13. Отключение запроса в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Delete(Employee employee) {
  // context.Set<SecondaryIdentity>().FirstOrDefault(id =>
  //   id.PrimarySSN == employee.SSN
  //   && id.PrimaryFirstName == employee.FirstName
  //   && id.PrimaryFamilyName == employee.FamilyName);
  context.Employees.Remove(employee);
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Запустите приложение, используя dotnet run, перейдите по ссылке http://localhost:5000 и щелкните на кнопке Delete. Свойства внешних ключей связанного объекта SecondaryIdentity установятся в null, как в предыдущем примере, но без необходимости в выполнении добавочного запроса в методе действия Delete().

Взятие под свой контроль операции удаления

Поведение Restrict указывает инфраструктуре Entity Framework Core и серверу баз данных о том, что вносить изменения в зависимые сущности не нужно. Выбрав такое поведение, вы несете полную ответственность за гарантирование, что операция удаления может быть выполнена без возникновения ошибки. Выберите поведение Restrict в классе контекста (листинг 22.14).

Листинг 22.14. Выбор поведения Restrict в файле AdvancedContext.cs из папки Models

```
...
modelBuilder.Entity<SecondaryIdentity>()
  .HasOne(s => s.PrimaryIdentity)
  .WithOne(e => e.OtherIdentity)
  .HasPrincipalKey<Employee>(e => new { e.SSN,
    e.FirstName, e.FamilyName })
  .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
    s.PrimaryFirstName, s.PrimaryFamilyName })
  .OnDelete(DeleteBehavior.Restrict);
...
```

Выполните в папке проекта AdvancedApp команды из листинга 22.15 для создания новой миграции и ее применения к БД.

Листинг 22.15. Создание и применение миграции к БД

```
dotnet ef migrations add RestrictDelete
dotnet ef database update
```

Воссоздание поведения каскадного удаления

Если вы хотите удалить связанные данные, тогда должны либо запросить у БД объекты, подлежащие удалению, либо обеспечить наличие в HTTP-запросе POST достаточного набора данных для создания объекта, который позволит инфраструктуре Entity Framework Core выполнить операцию удаления.

Добавьте в представление Index.cshtml скрытый элемент input, который помещает значение первичного ключа объекта SecondaryIdentity в запрос на удаление объекта Employee (см. листинг 22.12). Используйте это значение для сообщения инфраструктуре Entity Framework Core о необходимости включения связанного объекта в операцию удаления (листинг 22.16).

Листинг 22.16. Удаление связанных данных в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Delete(Employee employee) {
  if (employee.OtherIdentity != null) {
    context.Set<SecondaryIdentity>().Remove(employee.OtherIdentity);
  }
  context.Employees.Remove(employee);
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete (возможно, понадобится создать новые объекты, чтобы было что удалять). Просмотрите журнальные сообщения, сгенерированные приложением, и вы заметите две операции, которые удаляют данные из БД. Первая операция удаляет объект `SecondaryIdentity`:

```
...
DELETE FROM [SecondaryIdentity]
WHERE [Id] = @p0;
...
```

Удаление зависимой сущности подготавливает почву для второй операции, которая удаляет главную сущность:

```
...
DELETE FROM [Employees]
WHERE [SSN] = @p1 AND [FirstName] = @p2 AND [FamilyName] = @p3
  AND [RowVersion] = @p4;
...
```

Воссоздание поведения установки в `null`

При желании нести ответственность за установку свойств внешних ключей в `null` можете использовать метод `Attach()` для помещения объекта, созданного связывателем моделей MVC, под контроль отслеживания изменений и затем установить значения свойств внешних ключей, как показано в листинге 22.17.

На заметку! Прием работает только для необязательных отношений. Попытка установки в `null` свойства внешнего ключа для обязательного отношения приведет к ошибке.

Листинг 22.17. Установка свойств внешних ключей в файле `HomeController.cs` из папки `Controllers`

```
...
[HttpPost]
public IActionResult Delete(Employee employee) {
  if (employee.OtherIdentity != null) {
    SecondaryIdentity identity =
      context.Set<SecondaryIdentity>().Find(employee.OtherIdentity.Id);
```

```
    identity.PrimarySSN = null;
    identity.PrimaryFirstName = null;
    identity.PrimaryFamilyName = null;
  }
  employee.OtherIdentity = null;
  context.Employees.Remove(employee);
  context.SaveChanges();
  return RedirectToAction(nameof(Index));
}
...
```

Сначала у БД запрашиваются текущие значения внешних ключей, затем устанавливаются в `null` свойства внешних ключей объекта `SecondaryIdentity` и, наконец, устанавливается в `null` навигационное свойство `OtherIdentity` объекта `Employee`. Когда инфраструктура Entity Framework Core обновляет БД, она отправляет команды `UPDATE` и `DELETE`, которые подобны командам, применяемым поведением `ClientSetNull`.

Восстановление средства мягкого удаления

В заключение главы будет восстановлено средство мягкого удаления, а средство постоянного удаления перемещено в отдельный контроллер `Delete`.

Модифицируйте контроллер `Home`, чтобы метод действия `Delete()` выполнял мягкое удаление, как демонстрируется в листинге 22.18. Также измените запрос связанных данных в методе действия `Index()`, чтобы обращаться к объектам `SecondaryIdentity` через навигационные свойства объектов `Employee`. Это гарантирует обработку запроса фильтром запросов, так что объекты `SecondaryIdentity` со свойствами внешних ключей, равными `null`, отображаться не будут.

Листинг 22.18. Обновление методов действий в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System;
using System.Collections.Generic;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      IEnumerable<Employee> data = context.Employees
        .Include(e => e.OtherIdentity)
        .OrderByDescending(e => e.LastUpdated)
        .ToArray();
      ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
      return View(data);
    }
```

```
    public IActionResult Edit(string SSN, string firstName, string familyName)
    {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }
    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
      } else {
        Employee e = new Employee {
          SSN = employee.SSN, FirstName = employee.FirstName,
          FamilyName = employee.FamilyName,
          RowVersion = employee.RowVersion
        };
        context.Employees.Attach(e);
        e.Salary = employee.Salary;
        e.LastUpdated = DateTime.Now;
      }
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Employee employee) {
      context.Employees.Attach(employee);
      employee.SoftDeleted = true;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Фильтры запросов применяются только к запросам для классов, к которым они применены, т.е. в случае использования запросов, начинающихся с разных сторон отношения, могут быть получены несогласованные результаты. В методе действия `Index()` запрос для объектов `SecondaryIdentity` изменен так, чтобы он выбирал подмножество результатов из запроса для объектов `Employee`. Тем самым гарантируется, что пользователю будут отображаться только объекты `SecondaryIdentity`, связанные с объектами `Employee`, которые не были мягко удалены.

Совет. За счет применения метода `ToArray()` было обеспечено выполнение только одного запроса. Без него при перечислении представлением объектов `Employee` и `SecondaryIdentity` возникали бы дублированные запросы.

Добавьте кнопки в представление Index, используемое контроллером Delete, чтобы мягко удаленные объекты можно было удалять из БД, как по отдельности, так и массово (листинг 22.19).

Листинг 22.19. Добавление элементов в файле Index.cshtml из папки Views/Delete

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Deleted Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="4" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td class="text-right">
          <form method="post">
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="FirstName" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                asp-for="@e.RowVersion" />
            <input type="hidden" name="OtherIdentity.Id"
                 value="@e.OtherIdentity.Id" />
            <button asp-action="Restore" class="btn btn-sm btn-success">
              Restore
            </button>
            <button asp-action="Delete" class="btn btn-sm btn-danger">
              Delete
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <form method="post" asp-action="DeleteAll">
    <button type="submit" class="btn btn-danger">Delete All</button>
  </form>
</div>
```

Добавьте в контроллер `Delete` действия, которые соответствуют элементам, добавленным в листинге 22.19 (листинг 22.20).

Листинг 22.20. Добавление действий в файле `DeleteController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Collections.Generic;

namespace AdvancedApp.Controllers {

  public class DeleteController : Controller {
    private AdvancedContext context;

    public DeleteController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      return View(context.Employees.Where(e => e.SoftDeleted)
        .Include(e => e.OtherIdentity).IgnoreQueryFilters());
    }

    [HttpPost]
    public IActionResult Restore(Employee employee) {
      context.Employees.IgnoreQueryFilters()
        .First(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName).SoftDeleted = false;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult Delete(Employee e) {
      if (e.OtherIdentity != null) {
        context.Remove(e.OtherIdentity);
      }
      context.Employees.Remove(e);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public IActionResult DeleteAll() {
      IEnumerable<Employee> data = context.Employees
        .IgnoreQueryFilters()
        .Include(e => e.OtherIdentity)
        .Where(e => e.SoftDeleted).ToArray();
      context.RemoveRange(data.Select(e => e.OtherIdentity));
      context.RemoveRange(data);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Методы действий обязаны гарантировать, что объекты `SecondaryIdentity`, связанные с объектами `Employee`, также удаляются из БД, поскольку модель данных сконфигурирована с поведением удаления `Restrict`. Чтобы протестировать средство мягкого/жесткого удаления, запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и щелкните на кнопке Delete для одного или большего числа объектов `Employee`. Перейдите по ссылке `http://localhost:5000/delete`, где с помощью кнопок можно восстанавливать объекты, а также навсегда удалять одиночные или все мягко удаленные объекты (рис. 22.7).

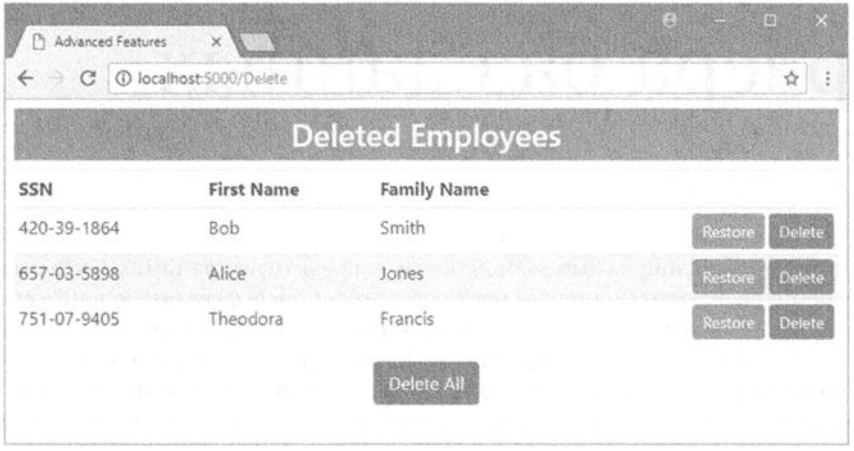

Рис. 22.7. Восстановление и завершение средства мягкого удаления

Резюме

В главе рассматривались различные линии поведения, которые инфраструктура Entity Framework Core поддерживает для удаления данных. Были показаны отличия между линиями поведения `Cascade` и `SetNull`/`ClientSetNull` и приведены объяснения, как взять под свой контроль процесс удаления, используя поведение `Restrict`. В заключение главы было восстановлено средство мягкого удаления и добавлена поддержка для удаления мягко удаленного объекта на постоянной основе. В следующей главе будут описаны средства, которые инфраструктура Entity Framework Core предлагает для работы с расширенными возможностями, предоставляемыми серверами баз данных.

ГЛАВА 23

Использование возможностей сервера баз данных

Несмотря на расширенные возможности, инфраструктура Entity Framework Core стремится охватить наиболее часто востребованные функциональные средства, предлагаемые серверами баз данных. Но принимая во внимание, что в ряде проектов таких средств может быть недостаточно, в Microsoft решили включить поддержку для работы непосредственно с сервером баз данных, которая может оказаться неоценимой при наличии в проекте специальных требований или при моделировании сложной БД.

В этой главе будет показано, как использовать SQL напрямую для доступа к функциональным средствам, которые не поддерживаются самой инфраструктурой Entity Framework Core, включая представления данных, хранимые процедуры и табличные функции. Также будет продемонстрирован целый диапазон возможностей, предлагаемых Entity Framework Core для работы со значениями данных, которые генерируются сервером баз данных. В табл. 23.1 приведены сведения, позволяющие поместить возможности сервера баз данных в контекст.

Внимание! Все функциональные возможности Entity Framework Core варьируются между серверами баз данных и пакетами поставщиков, что особенно касается примеров, рассмотренных в настоящей главе. Примеры были протестированы с применением SQL Server и стандартного поставщика БД от Microsoft. Если вы используете другой сервер баз данных или пакет поставщика, то возможно придется внести изменения в примеры.

Таблица 23.1. Помещение возможностей сервера баз данных в контекст

Вопрос	Ответ
Что это такое?	Серверы баз данных предлагают расширенные возможности, которые недоступны с применением обычных приемов Entity Framework Core. Инфраструктура Entity Framework Core содержит набор инструментов для работы с такими возможностями, который позволяет выйти за рамки стандартного набора средств, общих для всех серверов баз данных

Окончание табл. 23.1

Вопрос	Ответ
Чем они полезны?	Возможности сервера баз данных полезны для увеличения объема работы, предпринимаемой сервером баз данных от имени приложения, или когда нужна отдельная функция, к которой нельзя обратиться по-другому. В случае работы с существующей БД такие возможности могут понадобиться для извлечения и сохранения данных
Как они используются?	Возможности сервера баз данных доступны через комбинацию методов, включенных в запросы LINQ, и операторов Fluent API, которые конфигурируют модель данных
Существуют ли какие-то скрытые ловушки или ограничения?	Работа непосредственно с функциями базы данных затрудняет тестирование кода приложения и требует хорошего понимания расширенных возможностей сервера баз данных, а также кода SQL, требуемого для их применения
Существуют ли альтернативы?	Нет. Описанные в главе приемы являются единственным способом доступа к расширенным возможностям сервера баз данных

В табл. 23.2 приведена сводка по главе.

Таблица 23.2. Сводка по главе

Задача	Решение	Листинг
Выполнение SQL-команд	Используйте метод `FromSql()` или `ExecuteSqlCommand()`	23.1–23.18
Обновление объектов модели данных для отражения значений, сгенерированных сервером	Применяйте метод `HasDefaultValueSql()` или `HasSequence()` либо методы `ValueGeneratedOnXXX()`	23.19–23.25, 23.30–23.33
Обеспечение уникальности значений	Используйте метод `HasIndex()`	23.26–23.29

Подготовительные шаги

В главе продолжается работа с проектом `AdvancedApp`, который применялся, начиная с главы 19. Некоторые запросы не включали связанных данных, поэтому измените представление `Index`, используемое контроллером `Home`, как показано в листинге 23.1.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 23.1. Учет ранее отсутствующих связанных данных в файле `Index.cshtml` из папки `Views/Home`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                  asp-for="@e.RowVersion" />
            <input type="hidden" name="OtherIdentity.Id"
                  value="@e.OtherIdentity?.Id" />
            <button type="submit" asp-action="Delete" formmethod="post"
                   class="btn btn-sm btn-danger">
              Delete
            </button>
            <button type="submit" asp-action="Edit" formmethod="get"
                   class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
```

```
@if (ViewBag.Secondaries != null) {
  @Html.Partial("SecondaryIdentities")
}
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

В начале главы фильтр запросов не нужен, так что закомментируйте в классе контекста оператор Fluent API, который отфильтровывает мягко удаленные объекты (листинг 23.2).

Листинг 23.2. Отключение фильтра запросов в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      // modelBuilder.Entity<Employee>()
      //   .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Property(e => e.RowVersion).IsRowVersion();

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName })
        .OnDelete(DeleteBehavior.Restrict);

      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

В ряде примеров, приводимых в текущей главе, требуется оценка запросов на стороне клиента, поэтому закомментируйте в классе `Startup` оператор, который сообщает инфраструктуре Entity Framework Core о необходимости генерации исключения, когда часть запроса будет оцениваться в приложении (листинг 23.3).

Листинг 23.3. Запрет генерации исключения при оценке запросов на стороне клиента в файле `Startup.cs` из папки `AdvancedApp`

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

namespace AdvancedApp {
  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<AdvancedContext>(options =>
        options.UseSqlServer(conString));
          // .ConfigureWarnings(warning => warning.Throw(
          //  RelationalEventId.QueryClientEvaluationWarning)));
    }

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

Выполните в папке проекта `AdvancedApp` команды из листинга 23.4, чтобы удалить и воссоздать БД.

Листинг 23.4. Удаление и воссоздание БД

```
dotnet ef database drop --force
dotnet ef database update
```

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create (Создать) и сохраните три объекта `Employee`, используя значения из табл. 23.3.

Таблица 23.3. Значения данных для создания объектов в примере

SSN (**Номер карточки социального страхования**)	First Name (**Имя**)	Family Name (**Фамилия**)	Salary (**Оклад**)	Other Name (**Другое имя**)	In Active Use (**Активно используется**)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
657-03-5898	Alice	Jones	200000	Allie	Флажок отмечен
300-30-0522	Peter	Davies	180000	Pete	Флажок отмечен

Результат создания трех объектов `Employee` показан на рис. 23.1.

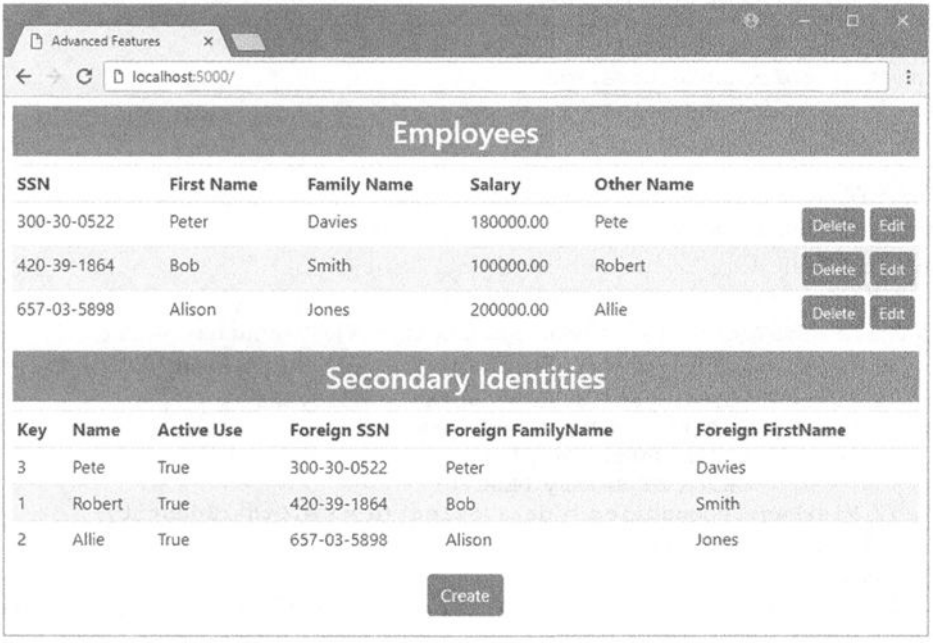

Рис. 23.1. Выполнение примера приложения

Использование SQL напрямую

Инфраструктура Entity Framework делает неплохую работу по предоставлению доступа к возможностям сервера баз данных, в которых нуждается большинство проектов, но каждый сервер баз данных обладает уникальным набором возможностей, взаимодействие с которыми означает работу непосредственно с кодом SQL. В последующих разделах будут объясняться разнообразные виды поддержки работы с SQL в инфраструктуре Entity Framework Core и затем демонстрироваться на конкретных примерах.

Внимание! Расширенные возможности должны применяться, только если требуемых результатов не удается добиться с использованием стандартных возможностей Entity Framework Core. Работа напрямую с SQL затрудняет тестирование и сопровождение приложения и может ограничить его взаимодействием лишь с одним сервером баз данных. Если вы имеете дело не с сервером SQL Server, тогда можете получить другие результаты в примерах этой главы.

Запрашивание с использованием SQL

Инфраструктура Entity Framework Core поддерживает метод `FromSql()`, предназначенный для запрашивания БД с применением SQL напрямую. В целях демонстрации измените метод действия `Index()` контроллера `Home`, чтобы он запрашивал БД, используя SQL (листинг 23.5).

Листинг 23.5. Запрашивание с применением SQL в файле `HomeController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System;
using System.Collections.Generic;

namespace AdvancedApp.Controllers {

  public class HomeController : Controller {
    private AdvancedContext context;

    public HomeController(AdvancedContext ctx) => context = ctx;

    public IActionResult Index() {
      IEnumerable<Employee> data = context.Employees
        .FromSql(@"SELECT * FROM Employees
                   WHERE SoftDeleted = 0
                   ORDER BY Salary DESC");
      // ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
      return View(data);
    }

    public IActionResult Edit(string SSN, string firstName, string familyName)
    {
      return View(string.IsNullOrWhiteSpace(SSN)
        ? new Employee() : context.Employees.Include(e => e.OtherIdentity)
          .First(e => e.SSN == SSN
            && e.FirstName == firstName
            && e.FamilyName == familyName));
    }

    [HttpPost]
    public IActionResult Update(Employee employee) {
      if (context.Employees.Count(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName) == 0) {
        context.Add(employee);
```

```
    } else {
      Employee e = new Employee {
        SSN = employee.SSN, FirstName = employee.FirstName,
        FamilyName = employee.FamilyName,
        RowVersion = employee.RowVersion
      };
      context.Employees.Attach(e);
      e.Salary = employee.Salary;
      e.LastUpdated = DateTime.Now;
    }
    context.SaveChanges();
    return RedirectToAction(nameof(Index));
  }

  [HttpPost]
  public IActionResult Delete(Employee employee) {
    context.Employees.Attach(employee);
    employee.SoftDeleted = true;
    context.SaveChanges();
    return RedirectToAction(nameof(Index));
  }
 }
}
```

Метод `FromSql()` используется для создания запроса, включающего низкоуровневый код SQL, и в приведенном примере из таблицы `Employees` извлекаются данные, которые не были мягко удалены, а результаты упорядочиваются по значению `Salary`.

На код SQL, который можно применять с методом `FromSql()`, налагается ряд ограничений. Самое важное ограничение состоит в том, что вы должны обеспечить указание столбцов для всех свойств объекта сущностного класса, который создаст инфраструктура Entity Framework Core, а столбцы должны иметь такие же имена, как у свойств. Запрашивать можно только специфический сущностный класс через его `DbSet<T>`, причем связанные данные не могут быть включены, поэтому классы, не входящие в модель данных, запрашивать нельзя. Наконец, инфраструктура Entity Framework Core не будет создавать объекты связанных классов, даже если в низкоуровневом коде SQL присутствует конструкция `JOIN`. (Другой прием для получения связанных данных описан в разделе "Составление сложных запросов" далее в главе.)

Чтобы просмотреть код SQL, отправляемый серверу баз данных, запустите приложение с использованием `dotnet run` и перейдите по ссылке `http://localhost:5000`. Заглянув в журнальные сообщения, которые сгенерированы приложением, вы увидите запрос, соответствующий низкоуровневому коду SQL из листинга 23.5:

```
...
SELECT * FROM Employees
WHERE SoftDeleted = 0
ORDER BY Salary DESC
...
```

Поскольку связанные данные из БД не извлекаются, пользователю будут отображены только детали объектов `Employee` (рис. 23.2).

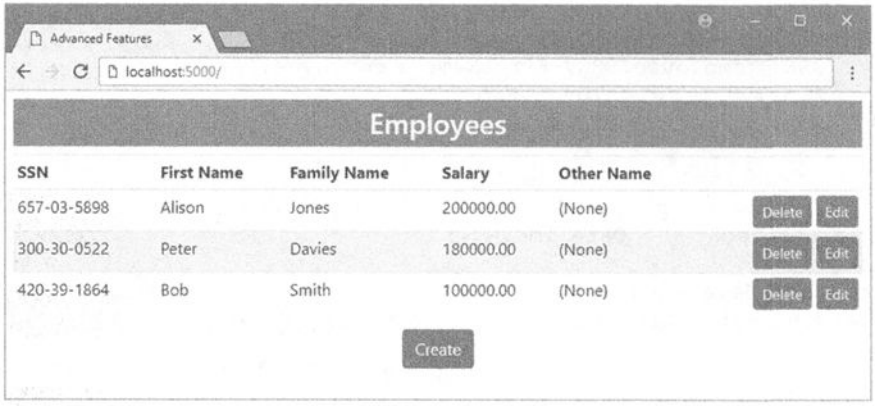

Рис. 23.2. Запрашивание БД с применением низкоуровневого кода SQL

Запрашивание с использованием параметров

Для предотвращения атак внедрением в SQL любой пользовательский ввод, включаемый в запрос SQL, потребуется параметризировать. Простейший способ предусматривает применение средства интерполяции строк, которое включает значения по именам в строку SQL и автоматически гарантирует их безопасную обработку. Измените запрос, выполняемый методом действия `Index()`, чтобы он принимал параметр, значение которого берется из HTTP-запроса, и использовал его для извлечения данных (листинг 23.6).

Листинг 23.6. Применение параметра запроса в файле `HomeController.cs` из папки `Controllers`

```
...
public IActionResult Index(decimal salary = 0) {
  IEnumerable<Employee> data = context.Employees
    .FromSql($@"SELECT * FROM Employees
                WHERE SoftDeleted = 0
                AND Salary > {salary}
                ORDER BY Salary DESC");
  // ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Снабжение строки префиксом в виде символа `$` делает возможным включение значений по именам, так что `{salary}` безопасным образом встраивает значение в запрос. Чтобы просмотреть сгенерированный SQL-запрос, запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`. Среди журнальных сообщений, сгенерированных приложением, вы заметите следующий запрос:

```
...
SELECT * FROM Employees
WHERE SoftDeleted = 0 AND Salary > @p0
ORDER BY Salary DESC
...
```

Инфраструктура Entity Framework Core отправила запрос, в котором вместо встраивания значения прямо в SQL-строку применяется безопасный параметр. Если вы перейдете по ссылке `http://localhost:5000?salary=100000`, то увидите, что данные фильтруются с отображением только тех объектов `Employee`, у которых значение свойства `Salary` превышает 100 000 (рис. 23.3).

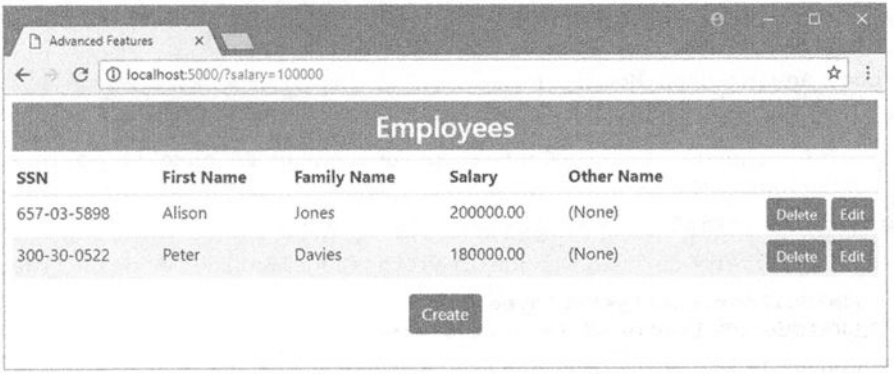

Рис. 23.3. Запрашивание с пользовательским вводом

Другие способы передачи параметров в запросы

Если вы не хотите использовать интерполяцию строк, тогда можете безопасно передавать параметры как аргументы методу `FromSql()`:

```
...
context.Employees.FromSql(@"SELECT * FROM Employees
                      WHERE SoftDeleted = 0 AND Salary > {0}
                      ORDER BY Salary DESC", salary);
...
```

Аргументы потребляются в порядке их указания, а ссылка на них в SQL-строке осуществляется по индексу, начиная с нуля. При работе со строкой, которая содержит имена параметров, требующихся запросу, для снабжения запроса значениями можете задействовать класс `SqlParameter`:

```
...
SqlParameter min = new SqlParameter("minSalary", salary);
IEnumerable<Employee> data = context.Employees
  .FromSql(@"SELECT * FROM Employees
           WHERE SoftDeleted = 0 AND Salary > @minSalary
           ORDER BY Salary DESC", min);
...
```

Оба приема производят запросы, совпадающие с запросами, которые получаются с помощью интерполяции строк, как было показано в листинге 23.6.

Составление сложных запросов

Если имеющийся сервер баз данных и пакет поставщика поддерживают это, тогда инфраструктура Entity Framework Core способна применять низкоуровневый код SQL в качестве фундамента для построения более сложного запроса, который формирует-

ся с использованием стандартных методов LINQ или других средств Entity Framework Core. Удалите комментарий с оператора Fluent API в классе контекста, активизировав фильтр запросов, который исключает мягко удаленные объекты (листинг 23.7).

Листинг 23.7. Включение фильтра запросов в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
      // ...для краткости остальные операторы не показаны...
    }
  }
}
```

Измените запрос, выполняемый методом действия `Index()`, чтобы включить связанные данные и упорядочить результаты с применением свойства `LastUpdated` (листинг 23.8).

Листинг 23.8. Составление сложного запроса в файле `HomeController.cs` из папки `Controllers`

```
...
public IActionResult Index(decimal salary = 0) {
  IEnumerable<Employee> data = context.Employees
    .FromSql($@"SELECT * FROM Employees
              WHERE SoftDeleted = 0
              AND Salary > {salary}")
    .Include(e => e.OtherIdentity)
    .OrderByDescending(e => e.Salary)
    .OrderByDescending(e => e.LastUpdated).ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Инфраструктура Entity Framework Core генерирует запрос, который объединяет низкоуровневый код SQL, переданный методу `FromSql()`, с дополнительным запросом, представленным методами LINQ. Для просмотра составного запроса запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000?salary=100000`; вы увидите в журнальных сообщениях такой запрос:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[LastUpdated],
  [e].[RowVersion], [e].[Salary], [e].[SoftDeleted], [e.OtherIdentity].[Id],
  [e.OtherIdentity].[InActiveUse], [e.OtherIdentity].[Name],
  [e.OtherIdentity].[PrimaryFamilyName],[e.OtherIdentity].[PrimaryFirstName],
  [e.OtherIdentity].[PrimarySSN]
FROM (
  SELECT * FROM Employees
  WHERE SoftDeleted = 0 AND Salary > @p0
) AS [e]
LEFT JOIN [SecondaryIdentity] AS [e.OtherIdentity] ON
  (([e].[SSN] = [e.OtherIdentity].[PrimarySSN])
  AND ([e].[FirstName] = [e.OtherIdentity].[PrimaryFirstName]))
  AND ([e].[FamilyName] = [e.OtherIdentity].[PrimaryFamilyName])
WHERE [e].[SoftDeleted] = 0
ORDER BY [e].[LastUpdated] DESC, [e].[Salary] DESC
...
```

Внутренняя часть запроса — это строка, переданная методу `FromSql()`, которая окружена внешним запросом. Внешний запрос применяет оператор `SELECT` для получения имен столбцов, которые необходимы инфраструктуре Entity Framework Core для создания объектов `Employee` и `SecondaryIdentity`, конструкцию `JOIN` для извлечения связанных данных, конструкцию `WHERE` для исключения мягко удаленных данных и конструкцию `ORDER BY` для сортировки данных по значению свойства `LastUpdated`.

Сочетание низкоуровневого кода SQL и стандартных методов LINQ может сделать запросы более управляемыми и тестируемыми, чем запросы, созданные целиком в низкоуровневом коде SQL. Тем не менее, такой прием накладывает ограничения на часть запроса, представленную низкоуровневым кодом SQL. Например, в листинге 23.8 обратите внимание на то, что конструкция `WHERE`, которая выбирает данные на основе значения `Salary`, перенесена в вызов LINQ-метода `OrderByDescending()`. Одно из ограничений, накладываемых на низкоуровневый код SQL в составных запросах, связано с тем, что конструкции `ORDER BY` использовать нельзя.

Использование низкоуровневого кода SQL для запрашивания с помощью хранимых процедур

Хранимые процедуры часто встречаются при работе с БД, которая предшествовала приложению ASP.NET Core MVC, либо при наличии специфических требований к производительности или управлению данными. Инфраструктура Entity Framework Core поддерживает запрашивание с помощью хранимой процедуры, хотя сервер баз данных или поставщик может накладывать ограничения на типы запросов, которые допускается делать.

В листинге 23.9 содержатся операторы SQL, требующиеся для создания хранимой процедуры, которая запрашивает у БД объекты `Employee` со значением `Salary`, превышающим указанную величину. Выберите пункт меню Tools⇨SQL Server⇨New Query (Сервис⇨SQL Server⇨Новый запрос) в среде Visual Studio, подключитесь к БД и выполните код SQL из листинга 23.9, чтобы создать хранимую процедуру.

Совет. Вы можете загрузить файл, который содержит SQL-операторы из листинга 23.9, из хранилища исходного кода для книги по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 23.9. Простая хранимая процедура

```
USE AdvancedDb
GO

DROP PROCEDURE IF EXISTS GetBySalary;
GO

CREATE PROCEDURE GetBySalary
  @SalaryFilter decimal
AS
  SELECT * from Employees
  WHERE Salary > @SalaryFilter AND SoftDeleted = 0
  ORDER BY Salary DESC
GO
```

Инфраструктура Entity Framework Core не может запрашивать с помощью хранимой процедуры с применением своих стандартных средств, что означает необходимость использования метода `FromSql()`. Измените запрос в методе действия `Index()` контроллера `Home`, чтобы он выполнял запрос к БД с помощью хранимой процедуры (листинг 23.10).

Листинг 23.10. Запрашивание с применением хранимой процедуры в файле `HomeController.cs` из папки `Controllers`

```
...
public IActionResult Index(decimal salary = 0) {
  IEnumerable<Employee> data = context.Employees
    .FromSql($"Execute GetBySalary @SalaryFilter = {salary}")
    .IgnoreQueryFilters();
  // ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
return View(data);
}
...
```

Типы запросов, которые можно выполнять с участием хранимой процедуры, ограничены. Скажем, не разрешено включать связанные данные, из-за чего был удален вызов `Include()`. Кроме того, удалены вызовы `OrderByDescending()` и добавлен вызов `IgnoreQueryFilters()`, чтобы предотвратить оценку на стороне клиента. Инфраструктура Entity Framework Core не способна формировать сложные запросы, использующие хранимые процедуры, поэтому придется либо обеспечить выполнение в хранимой процедуре всей требуемой фильтрации и обработки, либо принять необходимость в оценке на стороне клиента. Для тестирования запроса с хранимой процедурой запустите приложение с применением `dotnet run` и перейдите по ссылке `http://localhost:5000?salary=150000`. В журнальных сообщениях, сгенерированных приложением, вы увидите, что запрос использовал хранимую процедуру:

```
...
Execute GetBySalary @SalaryFilter = @p0
...
```

Поскольку связанные данные отсутствуют, результаты будут включать только данные `Employee` (рис. 23.4).

Рис. 23.4. Запрашивание с помощью хранимой процедуры

Составление сложных запросов с представлениями данных

Если вы можете влиять на структуру БД либо изменять ее, тогда инфраструктура Entity Framework Core предлагает более гибкую поддержку для представлений данных, которые являются виртуальными таблицами с содержимым, генерируемым запросами. В листинге 23.11 приведены SQL-операторы, предназначенные для создания простого представления данных, которое содержит все данные `Employee`, не подвергавшиеся мягкому удалению. Обычно представления данных оказываются более сложными или синтезируют данные за счет выполнения вычислений, но такого простого представления данных вполне достаточно для демонстрации их применения с инфраструктурой Entity Framework Core. Выберите пункт меню Tools⇨SQL Server⇨New Query в среде Visual Studio, подключитесь к БД и выполните код SQL из листинга 23.11, чтобы создать представление данных.

Листинг 23.11. Простое представление данных

```
USE AdvancedDb
GO

DROP VIEW IF EXISTS NotDeletedView
GO

CREATE VIEW NotDeletedView
AS
  SELECT * FROM Employees
  WHERE SoftDeleted = 0
GO
```

Для запрашивания представления данных модифицируйте метод действия `Index()` контроллера `Home`, как показано в листинге 23.12. Так как представление данных возвращает таблицу, инфраструктура Entity Framework Core имеет возможность сформировать запрос путем смешивания низкоуровневого кода SQL и стандартных методов запросов LINQ.

Листинг 23.12. Запрашивание представления данных в файле `HomeController.cs` из папки `Controllers`

```
...
public IActionResult Index(decimal salary = 0) {
  IEnumerable<Employee> data = context.Employees
    .FromSql($@"SELECT * from NotDeletedView
              WHERE Salary > {salary}")
    .Include(e => e.OtherIdentity)
    .OrderByDescending(e => e.Salary)
    .OrderByDescending(e => e.LastUpdated)
    .IgnoreQueryFilters()
    .ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Представление данных является источником данных в запросе и дополняется вызовами методов `Include()` и `OrderByDescending()`. Вызов метода `IgnoreQueryFilters()` был добавлен из-за того, что представление данных уже исключает мягко удаленные данные, делая фильтр запросов избыточным. Запустите приложение, перейдите по ссылке `http://localhost:5000` и найдите в журнальных сообщениях сформированный запрос, который был отправлен серверу баз данных:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName],
  [e].[LastUpdated],
  [e].[RowVersion], [e].[Salary], [e].[SoftDeleted], [e.OtherIdentity].[Id],
  [e.OtherIdentity].[InActiveUse], [e.OtherIdentity].[Name],
  [e.OtherIdentity].[PrimaryFamilyName],[e.OtherIdentity].[PrimaryFirstName],
  [e.OtherIdentity].[PrimarySSN]
FROM (SELECT * from NotDeletedView WHERE Salary > @p0) AS [e]
LEFT JOIN [SecondaryIdentity] AS [e.OtherIdentity]
  ON (([e].[SSN] = [e.OtherIdentity].[PrimarySSN])
  AND ([e].[FirstName] = [e.OtherIdentity].[PrimaryFirstName]))
  AND ([e].[FamilyName] = [e.OtherIdentity].[PrimaryFamilyName])
ORDER BY [e].[LastUpdated] DESC, [e].[Salary] DESC
...
```

Поскольку сформированный запрос включает связанные данные, пользователю отображаются объекты `Employee` и `SecondaryIdentity` (рис. 23.5).

Составление сложных запросов с применением табличных функций

Табличная функция похожа на нечто среднее между представлением данных и хранимой процедурой. Подобно хранимой процедуре табличная функция может принимать параметры, а в качестве результата она производит таблицу в точности как представление данных. При работе с инфраструктурой Entity Framework Core табличные функции часто могут заменять хранимые процедуры с тем преимуществом, что они позволяют составлять сложные запросы.

В листинге 23.13 приведены SQL-операторы, предназначенные для создания табличной функции, которая выполняет тот же самый запрос, что и хранимая процедура, созданная в листинге 23.9. Выберите пункт меню Tools⇨SQL Server⇨New Query в среде Visual Studio, подключитесь к БД и выполните код SQL из листинга 23.13, чтобы создать табличную функцию.

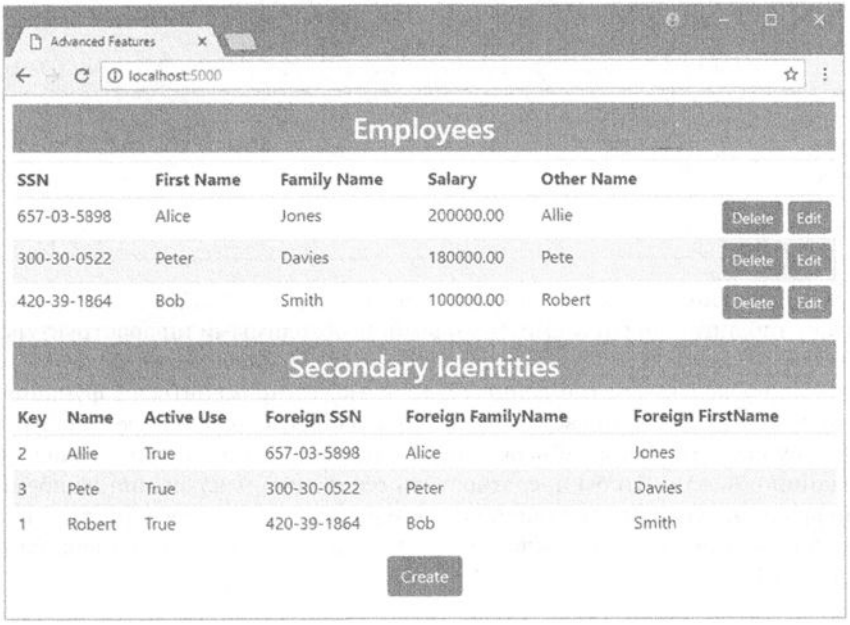

Рис. 23.5. Запрашивание с использованием представления данных

Совет. Инфраструктура Entity Framework Core также поддерживает скалярные функции, которые возвращают одиночное значение, а не таблицу результатов, производимую табличной функцией. Вместо использования кода SQL напрямую можно создать метод в классе контекста, и Entity Framework Core будет знать, что вызов этого метода в запросе должен привести к выполнению табличной функции. На момент написания книги данное средство работало только с ограниченным набором типов результатов. Подробные сведения ищите в описании метода `HasDbFunction()` из Fluent API.

Листинг 23.13. Табличная функция

```
USE AdvancedDb
GO

DROP FUNCTION IF EXISTS GetSalaryTable
GO

CREATE FUNCTION GetSalaryTable(@SalaryFilter decimal)
RETURNS @employeeInfo TABLE
(
  SSN nvarchar(450),
  FirstName nvarchar(450),
  FamilyName nvarchar(450),
  Salary decimal(8, 2),
  LastUpdated datetime2(7),
  SoftDeleted bit
) AS
```

```
  BEGIN
    INSERT INTO @employeeInfo
    SELECT SSN, FirstName, FamilyName, Salary, LastUpdated, SoftDeleted
    FROM Employees
    WHERE Salary > @SalaryFilter AND SoftDeleted = 0
      ORDER BY Salary DESC
    RETURN
  END
GO
```

Код SQL в листинге 23.13 создает функцию по имени `GetSalaryTable()`, которая возвращает таблицу с почти всеми столбцами, необходимыми инфраструктуре Entity Framework Core для создания объектов `Employee`. Исключением является столбец `RowVersion`, поскольку его тип данных SQL не может применяться в функциях.

Как отмечалось ранее, низкоуровневые SQL-запросы должны производить значения для всех свойств, которые необходимы инфраструктуре Entity Framework Core при создании объекта. Чтобы предотвратить генерацию исключения во время обработки инфраструктурой Entity Framework Core результатов запроса, настройте модель данных на игнорирование свойства `RowVersion` в классе `Employee`, как показано в листинге 23.14.

Внимание! Убедитесь в том, что вы понимаете последствия игнорирования свойств. В текущем примере игнорирование свойства `RowVersion` отключает защиту против параллельных обновлений, когда пользователь мягко удаляет объект. Остальные операции не затрагиваются, поскольку запросы, выполняемые другими методами действий в контроллере `Home`, не используют табличную функцию.

Листинг 23.14. Игнорирование свойства в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);
```

```
    modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
      .HasDefaultValue(new DateTime(2000, 1, 1));
    modelBuilder.Entity<Employee>()
      .Ignore(e => e.RowVersion);
      // .Property(e => e.RowVersion).IsRowVersion();
    modelBuilder.Entity<SecondaryIdentity>()
      .HasOne(s => s.PrimaryIdentity)
      .WithOne(e => e.OtherIdentity)
      .HasPrincipalKey<Employee>(e => new { e.SSN,
        e.FirstName, e.FamilyName })
      .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
        s.PrimaryFirstName, s.PrimaryFamilyName })
      .OnDelete(DeleteBehavior.Restrict);
    modelBuilder.Entity<SecondaryIdentity>()
      .Property(e => e.Name).HasMaxLength(100);
  }
 }
}
```

Для применения табличной функции модифицируйте запрос в методе действия `Index()` контроллера `Home` согласно листингу 23.15.

Листинг 23.15. Обращение к табличной функции в файле `HomeController.cs` из папки `Controllers`

```
...
public IActionResult Index(decimal salary = 0) {
  IEnumerable<Employee> data = context.Employees
    .FromSql($@"SELECT * from GetSalaryTable({salary})")
    .Include(e => e.OtherIdentity)
    // .OrderByDescending(e => e.Salary)
    .OrderByDescending(e => e.LastUpdated)
    .IgnoreQueryFilters()
    .ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Табличная функция используется в качестве цели в низкоуровневом SQL-запросе и принимает параметр, который применяется для фильтрации по значению `Salary`. Так как в случае использования табличных функций инфраструктура Entity Framework Core способна выполнять сложные запросы, появляется возможность включения связанных данных и упорядочения результатов. (Вызов метода `OrderByDescending()` для свойства `Salary` закомментирован из-за того, что функция уже сортирует данные по значению `Salary`.)

Чтобы увидеть эффект от применения табличной функции, запустите приложение, используя `dotnet run`, и перейдите по ссылке `http://localhost:5000`. Среди журнальных сообщений, сгенерированных приложением, вы увидите запрос, который инфраструктура Entity Framework Core сформировала для нацеливания на табличную функцию:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName],
  [e].[LastUpdated], [e].[Salary],
  [e].[SoftDeleted],[e.OtherIdentity].[Id],[e.OtherIdentity].[InActiveUse],
  [e.OtherIdentity].[Name], [e.OtherIdentity].[PrimaryFamilyName],
  [e.OtherIdentity].[PrimaryFirstName], [e.OtherIdentity].[PrimarySSN]
FROM (SELECT * from GetSalaryTable(@p0)) AS [e]
LEFT JOIN [SecondaryIdentity] AS [e.OtherIdentity]
  ON (([e].[SSN] = [e.OtherIdentity].[PrimarySSN])
  AND ([e].[FirstName] = [e.OtherIdentity].[PrimaryFirstName]))
  AND ([e].[FamilyName] = [e.OtherIdentity].[PrimaryFamilyName])
ORDER BY [e].[LastUpdated] DESC
...
```

Сформированный запрос следует той же структуре, которая демонстрировалась в предшествующих примерах. Поскольку запрос является составным, связанные данные загружаются и отображаются пользователю (рис. 23.6).

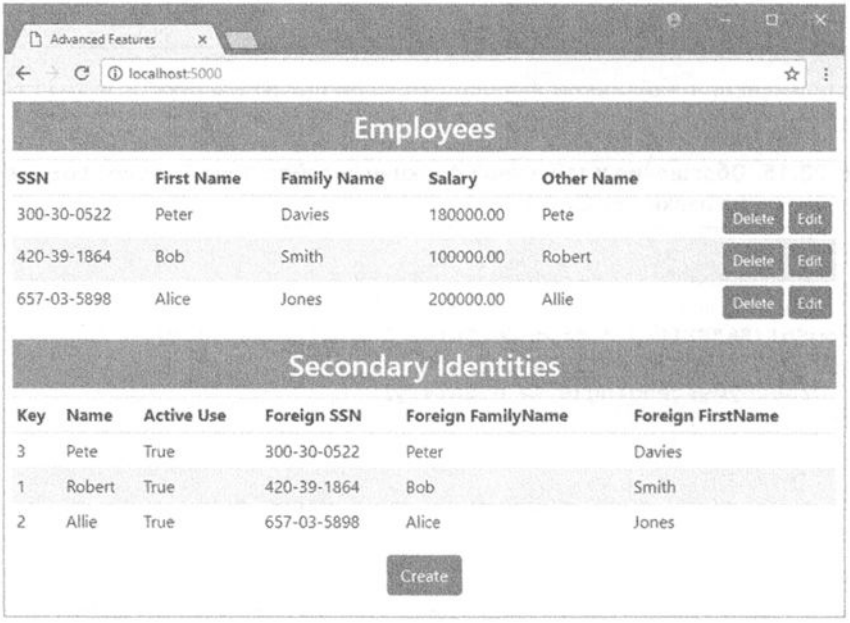

Рис. 23.6. Запрашивание с применением табличной функции

Вызов хранимых процедур или других операций

Не все хранимые процедуры используются для запрашивания данных, а это значит, что метод `FromSql()` может применяться не всегда. Хотя инфраструктура Entity Framework Core не открывает автоматический доступ к хранимым процедурам через класс контекста, ими по-прежнему можно пользоваться. В листинге 23.16 приведены SQL-операторы, предназначенные для создания двух хранимых процедур, которые восстанавливают мягко удаленные данные или удаляют их навсегда. Выберите пункт меню Tools⇨SQL Server⇨New Query в среде Visual Studio, подключитесь к БД и выполните код SQL из листинга 23.16.

Листинг 23.16. Две хранимые процедуры

```
USE AdvancedDb
GO

DROP PROCEDURE IF EXISTS RestoreSoftDelete
DROP PROCEDURE IF EXISTS PurgeSoftDelete
GO

CREATE PROCEDURE RestoreSoftDelete
AS
  BEGIN
    UPDATE Employees
    SET SoftDeleted = 0 WHERE SoftDeleted = 1
  END
GO

CREATE PROCEDURE PurgeSoftDelete
AS
  BEGIN
    DELETE from SecondaryIdentity WHERE Id IN
      (SELECT Id from Employees emp
      INNER JOIN SecondaryIdentity ident on ident.PrimarySSN = emp.SSN
      AND ident.PrimaryFirstName = emp.FirstName
      AND ident.PrimaryFamilyName = emp.FamilyName
      WHERE SoftDeleted = 1)
  END
  BEGIN
    DELETE FROM Employees
    WHERE SoftDeleted = 1
  END
```

Хранимые процедуры, не возвращающие данные, вызываются с применением метода `ExecuteSqlCommand()`, который используется в листинге 23.17 для обновления контроллера `Delete`.

Листинг 23.17. Вызов хранимых процедур в файле `DeleteController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Collections.Generic;

namespace AdvancedApp.Controllers {
  public class DeleteController : Controller {
    private AdvancedContext context;
    public DeleteController(AdvancedContext ctx) => context = ctx;
    public IActionResult Index() {
      return View(context.Employees.Where(e => e.SoftDeleted)
        .Include(e => e.OtherIdentity).IgnoreQueryFilters());
    }
    [HttpPost]
    public IActionResult Restore(Employee employee) {
```

```
      context.Employees.IgnoreQueryFilters()
        .First(e => e.SSN == employee.SSN
          && e.FirstName == employee.FirstName
          && e.FamilyName == employee.FamilyName).SoftDeleted = false;
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult Delete(Employee e) {
      if (e.OtherIdentity != null) {
        context.Remove(e.OtherIdentity);
      }
      context.Employees.Remove(e);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult DeleteAll() {
      context.Database.ExecuteSqlCommand("EXECUTE PurgeSoftDelete");
      return RedirectToAction(nameof(Index));
    }
    [HttpPost]
    public IActionResult RestoreAll() {
      context.Database.ExecuteSqlCommand("EXECUTE RestoreSoftDelete");
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Доступ к средствам SQL, которые не возвращают данные, осуществляются через метод `Database.ExecuteSqlCommand()` класса контекста, принимающий строку SQL (и необязательные параметры). В листинге 23.17 метод `ExecuteSqlCommand()` применялся для вызова хранимых процедур, определенных в листинге 23.16.

Метод действия `DeleteAll()` из листинга 23.17 уже можно использовать, поэтому добавьте в представление `Index` для контроллера `Delete` элемент HTML, нацеленный на действие `RestoreAll`.

Листинг 23.18. Добавление элемента в файле `Index.cshtml` из папки `Views/Delete`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Deleted Employees</h3>
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th></th>
    </tr>
  </thead>
```

```
  <tbody>
    <tr class="placeholder"><td colspan="4" class="text-center">No Data</td>
    </tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td class="text-right">
          <form method="post">
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="FirstName" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                   asp-for="@e.RowVersion" />
            <input type="hidden" name="OtherIdentity.Id"
                   value="@e.OtherIdentity.Id" />
            <button asp-action="Restore" class="btn btn-sm btn-success">
              Restore
            </button>
            <button asp-action="Delete" class="btn btn-sm btn-danger">
              Delete
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
<div class="text-center">
  <form method="post" asp-action="DeleteAll">
    <button type="submit" class="btn btn-danger">Delete All</button>
    <button type="submit" class="btn btn-success" asp-action="RestoreAll">
      Restore All
    </button>
  </form>
</div>
```

Чтобы удостовериться в вызове хранимых процедур, запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000` и с помощью кнопок Delete (Удалить) выполните мягкое удаление объектов `Employee`. Затем перейдите по ссылке `http://localhost:5000/delete` и воспользуйтесь кнопками Delete All (Удалить все) и Restore All (Восстановить все) для вызова хранимых процедур (рис. 23.7).

Использование значений, сгенерированных сервером

В главе 19 вы узнали, как сервер баз данных может брать на себя ответственность за генерацию уникальных ключей, но они не являются единственным видом значений, которые способен создавать сервер баз данных. В последующих разделах рассматриваются функциональные средства Entity Framework Core, которые поддерживают работу со значениями данных, сгенерированными сервером баз данных.

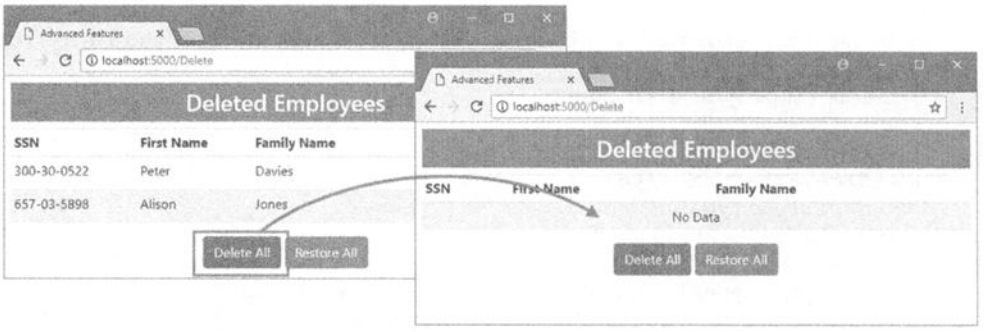

Рис. 23.7. Вызов хранимой процедуры

Использование стандартных значений, сгенерированных сервером

Во многих свойствах удобно применять метод `HasDefaultValue()` с фиксированным значением, как было показано в главе 21, поскольку это позволяет определить исходные данные для объектов, которые сохраняются в БД.

Однако в ряде проектов при сохранении нового объекта полезно запрашивать у сервера баз данных генерацию стандартного значения, несмотря на ограниченный диапазон стандартных значений, которые он может генерировать. В качестве примера добавьте в класс `Employee` новое свойство, как показано в листинге 23.19. Вместо добавления нового свойства для каждого вида значения, сгенерированного сервером, добавляется свойство, цель которого — исполнение роли заполнителя для простого отображения значений.

Листинг 23.19. Добавление свойства в файле `Employee.cs` из папки `Models`

```
using System;

namespace AdvancedApp.Models {

  public class Employee {
    private decimal databaseSalary;

    public long Id { get; set; }
    public string SSN { get; set; }
    public string FirstName { get; set; }
    public string FamilyName { get; set; }

    public decimal Salary {
      get => databaseSalary;
      set => databaseSalary = value;
    }

    public SecondaryIdentity OtherIdentity { get; set; }

    public bool SoftDeleted { get; set; } = false;

    public DateTime LastUpdated { get; set; }
    public byte[] RowVersion { get; set; }

    public string GeneratedValue { get; set; }
  }
}
```

Серверу баз данных сообщается о необходимости генерации стандартного значения для свойства с помощью метода `HasDefaultValueSql()` из Fluent API; он подобен методу `HasDefaultValue()`, но указывает SQL-выражение, которое должно быть выполнено для получения стандартного значения. В листинге 23.20 новое свойство конфигурируется с использованием метода `HasDefaultValueSql()`.

Листинг 23.20. Конфигурирование стандартного значения в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Ignore(e => e.RowVersion);
        // .Property(e => e.RowVersion).IsRowVersion();

      modelBuilder.Entity<Employee>().Property(e => e.GeneratedValue)
        .HasDefaultValueSql("GETDATE()");

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName })
        .OnDelete(DeleteBehavior.Restrict);

      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

В листинге 23.20 с применением SQL-функции GETDATE() получается отметка времени. Существуют ограничения на SQL-выражения, которые можно использовать для генерации стандартных значений, и потому диапазон значений лимитирован. Например, ссылаться на другие столбцы в той же таблице не разрешено, что имеет смысл, поскольку стандартное значение устанавливается при сохранении данных. В результате применение SQL-выражения для указания стандартных значений обычно включает вызов функций либо использование констант. Именно по этой причине большинство демонстраций стандартных значений полагаются на SQL-функцию GETDATE(), которая применялась в рассмотренном примере. Как будет показано в последующих разделах, доступны и более гибкие варианты.

Чтобы получить значение для свойства GeneratedValue, измените запрос в методе действия Index() контроллера Home; в нем больше не должна использоваться функция GetSalaryTable(), которая возвращает подмножество значений, требующихся инфраструктуре Entity Framework Core (листинг 23.21).

Листинг 23.21. Корректировка запроса в файле HomeController.cs из папки Controllers

```
...
public IActionResult Index() {
  IEnumerable<Employee> data = context.Employees
    // .FromSql($@"SELECT * from GetSalaryTable({salary})")
    .Include(e => e.OtherIdentity)
    // .OrderByDescending(e => e.Salary)
    .OrderByDescending(e => e.LastUpdated)
    .IgnoreQueryFilters()
    .ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Для конфигурирования БД нужна новая миграция. Выполните в папке проекта AdvancedApp команды из листинга 23.22, чтобы создать миграцию по имени GeneratedDefaultValue и применить ее к БД.

Листинг 23.22. Создание и применение миграции к БД

```
dotnet ef migrations add GeneratedDefaultValue
dotnet ef database update
```

Для отображения стандартного значения пользователю добавьте в представление Index, используемое контроллером Home (листинг 23.23).

Листинг 23.23. Добавление элементов в файле Index.cshtml из папки Views/Home

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
}
<h3 class="bg-info p-2 text-center text-white">Employees</h3>
```

```
<table class="table table-sm table-striped">
  <thead>
    <tr>
      <th>SSN</th>
      <th>First Name</th>
      <th>Family Name</th>
      <th>Salary</th>
      <th>Other Name</th>
      <th>Generated</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr class="placeholder"><td colspan="8" class="text-center">No Data</td>
</tr>
    @foreach (Employee e in Model) {
      <tr>
        <td>@e.SSN</td>
        <td>@e.FirstName</td>
        <td>@e.FamilyName</td>
        <td>@e.Salary</td>
        <td>@(e.OtherIdentity?.Name ?? "(None)")</td>
        <td>@e.GeneratedValue</td>
        <td class="text-right">
          <form>
            <input type="hidden" name="SSN" value="@e.SSN" />
            <input type="hidden" name="Firstname" value="@e.FirstName" />
            <input type="hidden" name="FamilyName" value="@e.FamilyName" />
            <input type="hidden" name="RowVersion"
                   asp-for="@e.RowVersion" />
            <input type="hidden" name="OtherIdentity.Id"
                    value="@e.OtherIdentity?.Id" />
            <button type="submit" asp-action="Delete" formmethod="post"
                     class="btn btn-sm btn-danger">
              Delete
            </button>
            <button type="submit" asp-action="Edit" formmethod="get"
                   class="btn btn-sm btn-primary">
              Edit
            </button>
          </form>
        </td>
      </tr>
    }
  </tbody>
</table>
@if (ViewBag.Secondaries != null) {
  @Html.Partial("SecondaryIdentities")
}
<div class="text-center">
  <a asp-action="Edit" class="btn btn-primary">Create</a>
</div>
```

Чтобы увидеть стандартное значение, сгенерированное сервером баз данных, запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create и сохраните новый объект. При сохранении новых данных сервер баз данных оценит SQL-выражение, использованное в листинге 23.21, и произведет результат, подобный показанному на рис. 23.8.

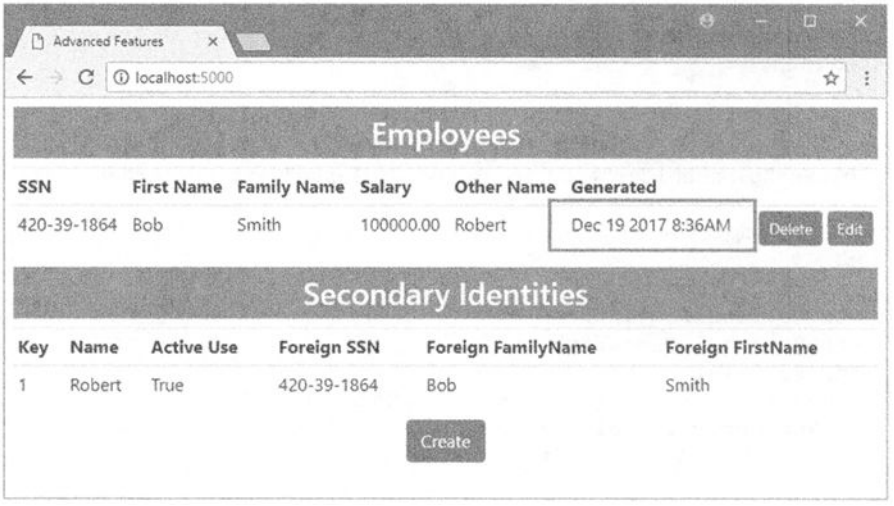

Рис. 23.8. Генерирование стандартного значения

Одним из следствий поручения серверу баз данных задачи генерации стандартных значений является то, что для выяснения, какое значение было присвоено, инфраструктура Entity Framework Core должна запрашивать БД. Если вы просмотрите журнальные сообщения из приложения, то обнаружите команду `INSERT`, которая применялась для сохранения нового объекта в БД:

```
...
INSERT INTO [Employees] ([SSN], [FirstName], [FamilyName], [Salary],
[SoftDeleted])
VALUES (@p0, @p1, @p2, @p3, @p4);
...
```

Немедленно после `INSERT` вы увидите запрос, используемый инфраструктурой Entity Framework Core для получения значений `GeneratedValue` и `LastUpdated`, которые были присвоены сервером баз данных:

```
...
SELECT [GeneratedValue], [LastUpdated]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [SSN] = @p0 AND [FirstName] = @p1 AND
[FamilyName] = @p2;
...
```

Инфраструктура Entity Framework Core применяет эти значения для обновления свойств сохраненного объекта, что гарантирует выполнение любых будущих операций над объектом с полным набором значений.

Встраивание последовательных значений

Один из способов увеличения гибкости стандартного значения, сгенерированного сервером, предусматривает встраивание последовательности, когда сервер баз данных будет генерировать уникальные значения по запросу. Оно похоже на средство генерации значений ключей, которое было описано в главе 19, но может применяться к любому свойству и включаться в сгенерированное значение.

Добавьте последовательность в модель данных и используйте ее как часть SQL-выражения, которое производит значение для свойства `GeneratedValue` (листинг 23.24).

Листинг 23.24. Применение последовательности в файле `AdvancedContext.cs` из папки `Models`

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {
  public class AdvancedContext : DbContext {
    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) {}

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Ignore(e => e.RowVersion);
        // .Property(e => e.RowVersion).IsRowVersion();

      modelBuilder.HasSequence<int>("ReferenceSequence")
        .StartsAt(100)
        .IncrementsBy(2);

      modelBuilder.Entity<Employee>().Property(e => e.GeneratedValue)
        .HasDefaultValueSql(@"'REFERENCE_'
          + CONVERT(varchar, NEXT VALUE FOR ReferenceSequence)");

      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName })
        .OnDelete(DeleteBehavior.Restrict);
```

```
        modelBuilder.Entity<SecondaryIdentity>()
          .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Последовательности создаются с использованием метода `HasSequence()`, в параметре типа которого указывается тип данных для последовательных значений, а в аргументе — имя, назначаемое последовательности. В качестве имени последовательности выбрано `ReferenceSequence`, но в реальном проекте должно применяться максимально значащее имя, потому что отдельно взятая последовательность может использоваться где угодно в БД.

Метод `HasSequence()` возвращает объект `SequenceBuilder`, который можно применять для конфигурирования последовательности с помощью методов, описанных в табл. 23.4.

Таблица 23.4. Методы конфигурирования последовательности

Имя	Описание
`StartsAt(значение)`	Используется для указания начального значения последовательности
`IncrementsBy(значение)`	Применяется для указания величины, на которую увеличивается генерируемое значение последовательности
`IsCyclic(циклы)`	Используется для указания, должна ли последовательность начинаться заново, когда достигнуто максимальное значение
`HasMax(значение)`	Применяется для указания максимального значения последовательности
`HasMin(значение)`	Используется для указания минимального значения последовательности

В листинге 23.24 посредством метода `StartsAt()` было указано начальное значение `100`, а с помощью метода `IncrementsBy()` задано увеличение на `2` генерируемого значения последовательности. Для применения последовательности измените выражение, передаваемое методу `HasDefaultValueSql()`, чтобы очередное значение последовательности преобразовывалось в строку и снабжалось префиксом `REFERENCE_`.

Применение последовательности требует новой миграции. Выполните в папке проекта `AdvancedApp` команды из листинга 23.25 для создания миграции по имени `Sequence` и ее применения к БД.

Листинг 23.25. Создание и применение миграции к БД

```
dotnet ef migrations add Sequence
dotnet ef database update
```

Чтобы увидеть эффект, запустите приложение, используя `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create и сохраните новый объект в БД. Когда отобразятся результаты, вы заметите, что при создании стандартного значения для нового объекта применялась последовательность (рис. 23.9).

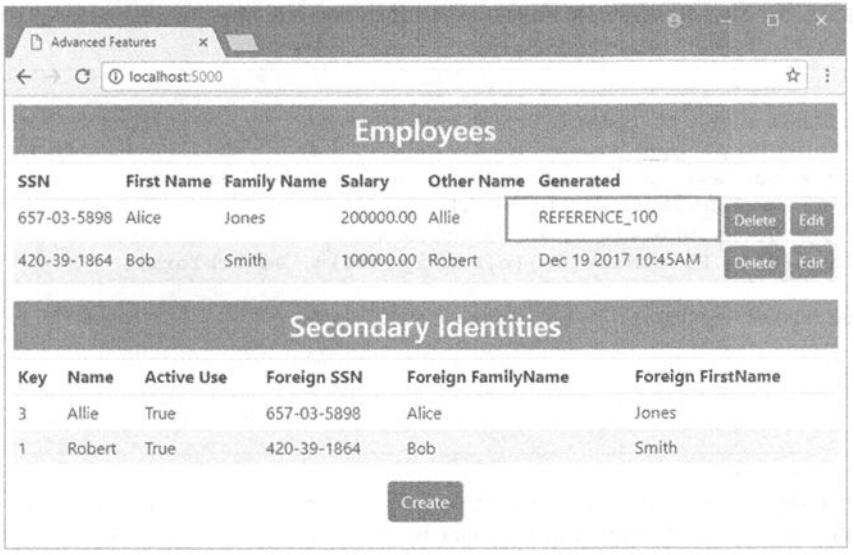

Рис. 23.9. Использование последовательности для генерации стандартных значений

Избегание ловушек, связанных с последовательностями

Есть две распространенные ловушки, в которые можно попасть при работе с последовательностями. Первая ловушка — излишняя ограниченность во время применения методов, описанных в табл. 23.4, что в результате дает пул доступных значений, который слишком мал, чтобы удовлетворять нуждам приложения. Спустя некоторое время после разработки все возможные значения израсходуются, и сервер баз данных начнет сообщать об ошибках.

Вторая ловушка — наиболее часто встречающаяся попытка исправления израсходованной последовательности, которая заключается в быстром применении метода `IsCyclic()`, чтобы начать последовательность заново. Значения в циклической последовательности не являются гарантированно уникальными, а потому одно и то же значение может быть присвоено несколько раз и вызовет проблемы в приложениях, где предполагается, что значения последовательности уникальны.

При создании последовательности выбирайте тип данных, который обеспечивает достаточно большой диапазон значений для разрабатываемого приложения. Если требуются уникальные значения, тогда обдумайте использование альтернативного ключа, как было описано в главе 19.

Вычисление значений в базе данных

Вычисляемый столбец — это такой столбец, который сервер баз данных вычисляет с применением значений, уже существующих в БД. Он может оказаться удобным способом генерации значений, которые иначе пришлось бы многократно вычислять при каждом выполнении запроса и возможно требовать оценки запроса на стороне клиента.

Измените запрос в методе действия `Index()` контроллера `Home`, чтобы пользователь мог применять поисковый термин для объединенных значений `FirstName` и `FamilyName`, как показано в листинге 23.26.

Листинг 23.26. Запрашивание с поисковым термином в файле HomeController.cs из папки Controllers

```
...
public IActionResult Index(string searchTerm) {
  IQueryable<Employee>
    query = context.Employees.Include(e => e.OtherIdentity);
  if (!string.IsNullOrEmpty(searchTerm)) {
    query = query.Where(e => EF.Functions
      .Like($"{e.FirstName[0]}{e.FamilyName}", searchTerm));
  }
  IEnumerable<Employee> data = query.ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Запрос берет первый символ свойства `FirstName`, связывает его со значением свойства `FamilyName` и выполняет поиск с использованием метода `Like()`. Запустите приложение посредством `dotnet run` и перейдите по ссылке `http://localhost:5000?searchTerm=%ajon%`. Поисковый термин `%ajon%` даст совпадение с объектом `Employee` вроде `Alice Jones`, хотя может понадобиться привести его в соответствие с хранящимися данными. Вы найдете следующее предупреждение в журнальных сообщениях, сгенерированных приложением:

```
...
The LINQ expression 'where __Functions_0.Like(Format("{0}{1}",
[e].FamilyName, Convert([e].FirstName.get_Chars(0), Object)),
__searchTerm_1)' could not be translated and will be evaluated locally
Выражение LINQ 'where __Functions_0.Like(Format("{0}{1}",
[e].FamilyName, Convert([e].FirstName.get_Chars(0), Object)),
__searchTerm_1)' не может быть транслировано и будет оцениваться локально
...
```

Как объяснялось в главе 20, метод `Like()` будет оцениваться на стороне клиента, если запрос не удается транслировать в SQL, что и произойдет в рассмотренном примере.

Вычисляемый столбец помогает избежать оценки на стороне клиента путем генерирования значений, которые можно запрашивать у БД. Измените конфигурацию свойства `GeneratedValue`, сделав его вычисляемым столбцом, который будет содержать объединенные значения `FirstName` и `FamilyName` (листинг 23.27).

Листинг 23.27. Определение вычисляемого столбца в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

    public AdvancedContext(DbContextOptions<AdvancedContext> options)
      : base(options) {}
```

```
    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);
      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });
      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);
      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));
      modelBuilder.Entity<Employee>()
        .Ignore(e => e.RowVersion);
        // .Property(e => e.RowVersion).IsRowVersion();
      modelBuilder.HasSequence<int>("ReferenceSequence")
        .StartsAt(100)
        .IncrementsBy(2);
      modelBuilder.Entity<Employee>().Property(e => e.GeneratedValue)
        // .HasDefaultValueSql(@"'REFERENCE_'
        //   + CONVERT(varchar, NEXT VALUE FOR ReferenceSequence)");
        .HasComputedColumnSql(@"SUBSTRING(FirstName, 1, 1)
                        + FamilyName PERSISTED");
      modelBuilder.Entity<Employee>().HasIndex(e => e.GeneratedValue);
      modelBuilder.Entity<SecondaryIdentity>()
        .HasOne(s => s.PrimaryIdentity)
        .WithOne(e => e.OtherIdentity)
        .HasPrincipalKey<Employee>(e => new { e.SSN,
          e.FirstName, e.FamilyName })
        .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
          s.PrimaryFirstName, s.PrimaryFamilyName })
        .OnDelete(DeleteBehavior.Restrict);
      modelBuilder.Entity<SecondaryIdentity>()
        .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Вычисляемые столбцы конфигурируются с применением метода `HasComputedColumnSql()`, который принимает SQL-выражение, используемое для генерации значений свойств. В этом случае SQL-выражение создает такое же сцепленное имя, как и ранее:

```
...
.HasComputedColumnSql(@"SUBSTRING(FirstName, 1, 1)
  + FamilyName PERSISTED");
...
```

Ключевое слово PERSISTED сообщает серверу баз данных о необходимости сохранить значения в БД на постоянной основе, а не генерировать их для каждого запроса. Кроме того, с помощью метода HasIndex() из Fluent API для свойства GeneratedValue создан индекс; индекс по вычисляемым столбцам не обязателен, но его наличие увеличит эффективность поиска.

Добавление вычисляемого столбца требует обновления БД. Выполните в папке проекта AdvancedApp команды, приведенные в листинге 23.28, чтобы создать новую миграцию и применить ее к БД.

Совет. Сервер баз данных будет автоматически вычислять заново значение вычисляемого столбца, когда изменяется любое значение, от которого он зависит. В текущем примере свойство SearchName будет вычисляться заново, если изменяется значение FirstName или FamilyName.

Листинг 23.28. Создание и применение миграции к БД

```
dotnet ef migrations add ComputedColumn
dotnet ef database update
```

Запрашивание с использованием вычисляемого столбца

После определения вычисляемый столбец можно применять в запросе подобно любому другому столбцу. Модифицируйте метод действия Index() контроллера Home, чтобы вызвать метод Like() на свойстве, которое соответствует вычисляемому столбцу (листинге 23.29).

Листинг 23.29. Использование вычисляемого столбца в файле HomeController.cs из папки Controllers

```
...
public IActionResult Index(string searchTerm) {
  IQueryable<Employee>
    query = context.Employees.Include(e => e.OtherIdentity);
  if (!string.IsNullOrEmpty(searchTerm)) {
    query = query.Where(e => EF.Functions.Like(e.GeneratedValue, searchTerm));
  }
  IEnumerable<Employee> data = query.ToArray();
  ViewBag.Secondaries = data.Select(e => e.OtherIdentity);
  return View(data);
}
...
```

Запустите приложение, используя dotnet run, и перейдите по ссылке http://localhost:5000?searchTerm=%ajon%; вы увидите результаты, показанные на рис. 23.10, хотя может понадобиться привести поисковый термин в соответствие с данными, хранящимися в БД. Если вы просмотрите журнальные сообщения, сгенерированные приложением, то заметите, что операция LIKE выполняется как часть SQL-запроса:

```
...
SELECT [e].[SSN], [e].[FirstName], [e].[FamilyName], [e].[GeneratedValue],
  [e].[LastUpdated], [e].[Salary], [e].[SoftDeleted], [e.OtherIdentity].[Id],
  [e.OtherIdentity].[InActiveUse], [e.OtherIdentity].[Name],
  [e.OtherIdentity].[PrimaryFamilyName],[e.OtherIdentity].[PrimaryFirstName],
  [e.OtherIdentity].[PrimarySSN]
FROM [Employees] AS [e]
LEFT JOIN [SecondaryIdentity] AS [e.OtherIdentity]
  ON (([e].[SSN] = [e.OtherIdentity].[PrimarySSN])
     AND ([e].[FirstName] = [e.OtherIdentity].[PrimaryFirstName]))
     AND ([e].[FamilyName] = [e.OtherIdentity].[PrimaryFamilyName])
WHERE ([e].[SoftDeleted] = 0) AND [e].[GeneratedValue]
LIKE @__searchTerm_1
...
```

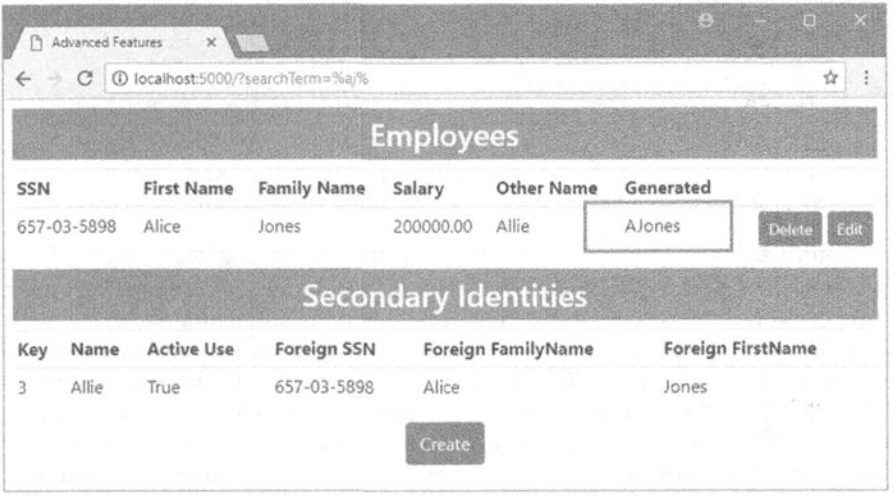

Рис. 23.10. Запрашивание с применением вычисляемого столбца

Моделирование автоматически генерируемых значений

В ряде проектов вы можете иметь дело со значениями, которые генерируются сервером баз данных, но не сконфигурированы в БД. Чаще так происходит из-за наличия в БД триггеров, генерирующих значения автоматически при вставке или обновлении данных в БД.

В качестве подготовительных шагов выполните команды из листинга 23.30, находясь в папке проекта AdvancedApp. Эти команды удаляют из проекта самую последнюю миграцию и воссоздают БД, что позволяет избежать проблемы, когда инфраструктура Entity Framework Core пытается удалить столбец, пока он еще используется индексом.

Листинг 23.30. Удаление миграции и переустановка БД

```
dotnet ef database drop --force
dotnet ef migrations remove --force
dotnet ef database update
```

В листинге 23.31 содержатся SQL-операторы, предназначенные для создания простого триггера, который обновляет свойство `GeneratedValue` при сохранении или обновлении объекта `Employee` в БД. Выберите пункт меню Tools⇨SQL Server⇨New Query в среде Visual Studio, подключитесь к БД и выполните код SQL из листинга 23.31, чтобы создать триггер.

Листинг 23.31. Создание триггера в БД

```
USE AdvancedDb
GO
DROP TRIGGER IF EXISTS GeneratedValueTrigger
GO
CREATE TRIGGER GeneratedValueTrigger ON Employees
  AFTER INSERT, UPDATE
AS
BEGIN
  DECLARE @Salary decimal(8,0), @SSN nvarchar(450),
        @First nvarchar(450), @Family nvarchar(450)
  SELECT @Salary = INSERTED.Salary, @SSN = INSERTED.SSN,
        @First = INSERTED.FirstName, @Family = INSERTED.FamilyName
  FROM INSERTED
  UPDATE dbo.Employees SET GeneratedValue = FLOOR(@Salary / 2)
  WHERE SSN = @SSN AND FirstName = @First AND FamilyName = @Family
END
```

Триггер выполняется при обновлении строки в таблице `Employee` и устанавливает значение свойства `GeneratedValue` равным половине значения `Salary`.

Инфраструктура Entity Framework Core не располагает поддержкой настройки средств, подобных триггерам в БД, но вы можете обеспечить проверку для получения значений, которые производит сервер баз данных, чтобы последующие операции, выполняемые с участием тех же самых объектов, не имели дело с неполными или устаревшими данными. Сообщить инфраструктуре Entity Framework Core о том, как генерируются значения для свойства, можно с применением четырех методов Fluent API, которые описаны в табл. 23.5.

Таблица 23.5. Методы Fluent API, предназначенные для генерирования значений свойств

Имя	Описание
`ValueGeneratedNever()`	Указывает инфраструктуре Entity Framework Core, что значения для свойства генерироваться не будут; является стандартным поведением
`ValueGeneratedOnAdd()`	Сообщает инфраструктуре Entity Framework Core о том, что значения для свойства будут генерироваться при сохранении нового объекта в БД
`ValueGeneratedOnUpdate()`	Указывает инфраструктуре Entity Framework Core, что значения для свойства будут генерироваться при обновлении существующего объекта
`ValueGeneratedOnAddOrUpdate()`	Сообщает инфраструктуре Entity Framework Core о том, что значения для свойства будут генерироваться при сохранении нового объекта в БД или при обновлении существующего объекта

Добавленный в БД триггер будет генерировать значение для столбца GeneratedValue при вставке или обновлении строки в таблице SecondaryIdentity. Для настройки модели данных измените конфигурацию свойства GenerateValue, удалив вычисляемый столбец и сообщив инфраструктуре Entity Framework Core о необходимости запрашивания сгенерированных значений при вставке или обновлении данных (листинг 23.32).

Ограничения методов для генерирования значений

Методы, описанные в табл. 23.5, не конфигурируют БД для генерации значений, а лишь сообщают инфраструктуре Entity Framework Core, когда она должна запрашивать сгенерированные значения, которые уже сконфигурированы. Таким образом, эти методы полезны для моделирования существующих БД, но не действуют, когда используются в проекте "сначала код".

Даже в проекте "сначала БД" методы из табл. 23.5 полезны, только если нужно делать последовательность запросов с применением того же самого объекта контекста БД, как в случае средства исправления. Дополнительные запросы, которые инфраструктура Entity Framework Core производит при использовании методов из табл. 23.5, будут бесполезными, если после выполнения операции создания или обновления контекст и кешированные данные отбрасываются, что происходит в большинстве приложений ASP.NET Core MVC.

Листинг 23.32. Конфигурирование свойства в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

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

    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {

      modelBuilder.Entity<Employee>()
        .HasQueryFilter(e => !e.SoftDeleted);

      modelBuilder.Entity<Employee>().Ignore(e => e.Id);
      modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.SSN, e.FirstName, e.FamilyName });

      modelBuilder.Entity<Employee>()
        .Property(e => e.Salary).HasColumnType("decimal(8,2)")
        .HasField("databaseSalary")
        .UsePropertyAccessMode(PropertyAccessMode.Field);

      modelBuilder.Entity<Employee>().Property<DateTime>("LastUpdated")
        .HasDefaultValue(new DateTime(2000, 1, 1));

      modelBuilder.Entity<Employee>()
        .Ignore(e => e.RowVersion);
        // .Property(e => e.RowVersion).IsRowVersion();
```

```
        modelBuilder.HasSequence<int>("ReferenceSequence")
          .StartsAt(100)
          .IncrementsBy(2);

        modelBuilder.Entity<Employee>().Property(e => e.GeneratedValue)
          .ValueGeneratedOnAddOrUpdate();

        modelBuilder.Entity<SecondaryIdentity>()
          .HasOne(s => s.PrimaryIdentity)
          .WithOne(e => e.OtherIdentity)
          .HasPrincipalKey<Employee>(e => new { e.SSN,
            e.FirstName, e.FamilyName })
          .HasForeignKey<SecondaryIdentity>(s => new { s.PrimarySSN,
            s.PrimaryFirstName, s.PrimaryFamilyName })
          .OnDelete(DeleteBehavior.Restrict);

        modelBuilder.Entity<SecondaryIdentity>()
          .Property(e => e.Name).HasMaxLength(100);
    }
  }
}
```

Изменение конфигурации свойства требует обновления БД. Выполните в папке проекта AdvancedApp команды из листинга 23.33, чтобы создать и применить миграцию к БД.

Листинг 23.33. Создание и применение миграции к БД

```
dotnet ef migrations add AutomaticallyGenerated
dotnet ef database update
```

Запустите приложение, используя dotnet run, перейдите по ссылке http://localhost:5000, щелкните на кнопке Create и сохраните новый объект Employee. При сохранении данных триггер установит значение столбца GeneratedValue в новой строке таблицы БД, давая результат, который показан на рис. 23.11.

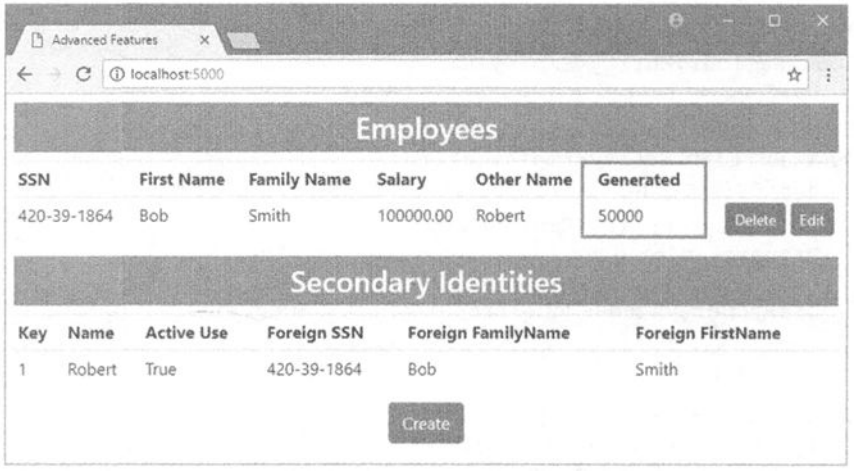

Рис. 23.11. Запрашивание автоматически сгенерированного значения

Тем не менее, в рассмотренном примере важно то, что инфраструктура Entity Framework Core будет запрашивать GeneratedValue после выполнения команды INSERT или UPDATE. В журнальных сообщениях, выпущенных приложением, можно обнаружить оператор INSERT, который применялся для сохранения созданного объекта:

```
...
INSERT INTO [Employees] ([SSN], [FirstName], [FamilyName], [Salary],
[SoftDeleted])
VALUES (@p0, @p1, @p2, @p3, @p4);
...
```

Непосредственно за этой командой находится запрос, который извлекает значения свойств, генерируемые сервером баз данных:

```
...
SELECT [GeneratedValue], [LastUpdated]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [SSN] = @p0 AND [FirstName] = @p1 AND
[FamilyName] = @p2;
...
```

Моделирование сгенерированных значений с использованием атрибутов

Если вы предпочитаете не иметь дело с Fluent API, тогда можете сообщить инфраструктуре Entity Framework Core о свойствах, значения которых генерируются сервером баз данных, с помощью атрибута DatabaseGenerated, принимающего в качестве аргумента значение перечисления DatabaseGeneratedOption:

```
...
[DatabaseGenerated(Computed)]
public string GeneratedValue { get; set; }
...
```

В перечислении DatabaseGeneratedOption определены три значения. Вариант None указывает, что значение генерироваться не будет, Identity сообщает о том, что значение генерируется при сохранении нового объекта, а Computed указывает инфраструктуре Entity Framework Core, что значение генерируется, когда сохраняется новый или обновляется существующий объект.

Резюме

В главе рассматривались средства Entity Framework Core, предназначенные для работы непосредственно с кодом SQL. Было показано, как включать код SQL напрямую в запросы, и каким образом создавать сложные запросы с применением различных возможностей сервера баз данных, в числе которых представления данных, хранимые процедуры и табличные функции. Кроме того, обсуждались способы, которыми сервер баз данных способен генерировать значения, и как эти значения можно встраивать в приложение. В следующей главе будет описана поддержка транзакций в инфраструктуре Entity Framework Core.

ГЛАВА 24

Использование транзакций

В этой главе будет описан способ, которым инфраструктура Entity Framework Core поддерживает транзакции, используемые для того, чтобы гарантировать выполнение множества операций как одной единицы работы. Если все операции могут быть выполнены без возникновения проблем, тогда изменения применяются к БД, что называется *фиксацией* транзакции. Если одна или большее число операций терпят неудачу, то БД не модифицируется и изменения отменяются, что называется *откатом* транзакции. (Приведенные объяснения несколько упрощены, т.к. транзакции могут быть более сложными, но они отражают суть работы транзакций.)

Сначала рассматривается стандартное поведение инфраструктуры Entity Framework Core, а затем будет показано, как взять транзакции под свой контроль, в том числе полностью отключить их. В табл. 24.1 приведены сведения, позволяющие поместить транзакции в контекст.

На заметку! Транзакции поддерживаются не всеми серверами баз данных. Примеры в главе рассчитаны на SQL Server, но если вы используете другой сервер баз данных, то можете столкнуться с отличающимся поведением.

Таблица 24.1. Помещение транзакций в контекст

Вопрос	Ответ
Что это такое?	Транзакции позволяют группировать изменения вместе, чтобы все они применялись в случае успеха, и ни одно изменение не применялось в случае неудачи любого из них
Чем они полезны?	Транзакции делают возможным группирование связанных операций для обеспечения согласованности в данных, используемых приложением, в дополнение к ссылочной целостности, которую поддерживает сервер баз данных
Как они используются?	По умолчанию транзакции включены, но доступен также API-интерфейс для конфигурирования способа применения транзакций
Существуют ли какие-то скрытые ловушки или ограничения?	Транзакции могут влиять на производительность и возможно возникновение взаимных блокировок из-за того, что обновления должны ставиться в очередь при обработке транзакций
Существуют ли альтернативы?	Приложения могут вносить корректирующие обновления для эмуляции эффекта отката транзакции, но делать это правильно нелегко

В табл. 24.2 приведена сводка по главе.

Таблица 24.2. Сводка по главе

Задача	Решение	Листинг
Обособленное выполнение обновлений	Вызывайте метод `SaveChanges()` после каждого обновления	24.5
Отключение автоматических транзакций	Используйте свойство `AutoTransactionsEnabled`	24.6, 24.7
Явное применение транзакций	Используйте методы `BeginTransaction()`, `CommitTransaction()` и `RollbackTransaction()`	24.8, 24.9
Указание уровня изоляции транзакций	Применяйте перечисление `IsolationLevel`	24.10

Подготовительные шаги

В главе продолжается работа с проектом `AdvancedApp`, который применялся, начиная с главы 19. В главе 23 с использованием низкоуровневого кода SQL были созданы средства вроде хранимых процедур и представлений данных, которые больше не нужны. Выполните в папке проекта `AdvancedApp` команды из листинга 24.1 для удаления и воссоздания БД.

Листинг 24.1. Переустановка БД

```
dotnet ef database drop --force
dotnet ef database update
```

В качестве подготовительных шагов добавьте в папку `Controllers` файл класса по имени `MultiController.cs` с содержимым, приведенным в листинге 24.2.

Листинг 24.2. Содержимое файла `MultiController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      return View("EditAll", context.Employees);
    }
```

```
    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      context.UpdateRange(employees);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы снабдить новый контроллер представлением, создайте папку `Views/Multi` и добавьте в нее файл по имени `EditAll.cshtml` с содержимым из листинга 24.3.

Совет. Если вы не хотите повторять процесс построения проекта примера, тогда можете загрузить все необходимые файлы из хранилища исходного кода для книги, доступного по ссылке `https://github.com/apress/pro-ef-core-2-for-asp.net-core-mvc`.

Листинг 24.3. Содержимое файла `EditAll.cshtml` из папки `Views/Multi`

```
@model IEnumerable<Employee>
@{
  ViewData["Title"] = "Advanced Features";
  Layout = "_Layout";
  int counter = 0;
}

<h4 class="bg-info p-2 text-center text-white">
  Edit All
</h4>

<form asp-action="UpdateAll" method="post">
  <div class="container">
    @foreach (Employee e in Model) {
      <div class="form-row">
        <div class="col">
          <input class="form-control" name="Employees[@counter].SSN"
                 value="@e.SSN" readonly />
        </div>
        <div class="col">
          <input class="form-control" name="Employees[@counter].FirstName"
                 value="@e.FirstName" readonly />
        </div>
        <div class="col">
          <input class="form-control" name="Employees[@counter].FamilyName"
                 value="@e.FamilyName" readonly />
        </div>
        <div class="col">
          <input class="form-control" name="Employees[@counter].Salary"
                  value="@e.Salary" />
        </div>
      </div>
      counter++;
    }
  </div>
```

```
  <div class="text-center m-2">
    <button type="submit" class="btn btn-primary">Save All</button>
    <a class="btn btn-secondary" asp-action="Index"
       asp-controller="Home">
      Cancel
    </a>
  </div>
</form>
```

Запустите приложение с применением `dotnet run`, перейдите по ссылке `http://localhost:5000`, щелкните на кнопке Create (Создать) и сохраните три объекта `Employee`, используя значения из табл. 24.3.

Таблица 24.3. Значения данных для создания объектов в примере

SSN (Номер карточки социального страхования)	First Name (Имя)	Family Name (Фамилия)	Salary (Оклад)	Other Name (Другое имя)	In Active Use (Активно используется)
420-39-1864	Bob	Smith	100000	Robert	Флажок отмечен
657-03-5898	Alice	Jones	200000	Allie	Флажок отмечен
300-30-0522	Peter	Davies	180000	Pete	Флажок отмечен

Результат создания трех объектов `Employee` показан на рис. 24.1.

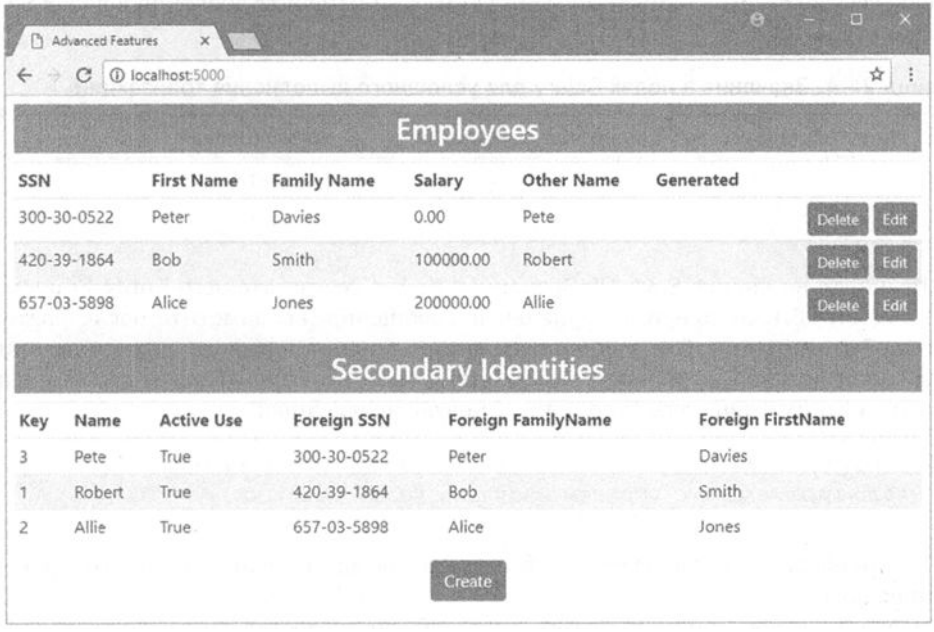

Рис. 24.1. Выполнение примера приложения

Стандартное поведение

Транзакции являются настолько фундаментальной частью работы с БД, что Entity Framework Core применяет их автоматически. Для прояснения особенностей использования транзакций измените конфигурацию системы ведения журналов, чтобы инфраструктура Entity Framework Core выдавала более подробные сообщения (листинг 24.4).

Листинг 24.4. Изменение уровня ведения журналов в файле `appsettings.json` из папки `AdvancedApp`

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

Добавленная запись требует сообщений уровня `Debug` от классов, которые обрабатывают транзакции Entity Framework Core. Чтобы увидеть, как транзакции применяются по умолчанию, запустите приложение с использованием `dotnet run`, перейдите по ссылке `http://localhost:5000/multi` и измените значения в полях Salary согласно табл. 24.4.

Таблица 24.4. Значения в полях Salary для успешного выполнения транзакции

First Name (Имя)	Family Name (Фамилия)	Salary (Оклад)
Bob	Smith	150000
Alice	Jones	250000

Щелкните на кнопке Save All (Сохранить все), в результате чего Entity Framework Core обновит БД. Заглянув в журнальные сообщения, вы заметите последовательность событий (хотя найти индивидуальные сообщения в выводе, сгенерированном настройкой ведения журналов `Debug`, может быть нелегко). Первым делом инфраструктура Entity Framework Core создает новую транзакцию:

```
...
Beginning transaction with isolation level 'ReadCommitted'.
Начало транзакции с уровнем изоляции ReadCommitted.
...
```

Уровни изоляции объясняются позже в главе, но здесь важно понять, что транзакция началась.

Инфраструктуре Entity Framework Core не были предоставлены исходные данные для обнаружения изменений, поэтому обновляются все объекты, давая в результате

три оператора UPDATE, за каждым из которых следует оператор SELECT, выясняющий сгенерированные сервером баз данных значения:

```
...
UPDATE [Employees] SET [LastUpdated] = @p0, [Salary] = @p1,
[SoftDeleted] = @p2
WHERE [SSN] = @p3 AND [FirstName] = @p4 AND [FamilyName] = @p5;
SELECT [GeneratedValue]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [SSN] = @p3 AND [FirstName] = @p4 AND
[FamilyName] = @p5;

UPDATE [Employees] SET [LastUpdated] = @p6, [Salary] = @p7,
[SoftDeleted] = @p8
WHERE [SSN] = @p9 AND [FirstName] = @p10 AND [FamilyName] = @p11;
SELECT [GeneratedValue]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [SSN] = @p9 AND [FirstName] = @p10 AND
[FamilyName] = @p11;

UPDATE [Employees] SET [LastUpdated] = @p12, [Salary] = @p13,
[SoftDeleted] = @p14
WHERE [SSN] = @p15 AND [FirstName] = @p16 AND [FamilyName] = @p17;
SELECT [GeneratedValue]
FROM [Employees]
WHERE @@ROWCOUNT = 1 AND [SSN] = @p15 AND [FirstName] = @p16 AND
[FamilyName] = @p17;
...
```

При выполнении показанных команд сервер баз данных не сообщал об ошибках, и у инфраструктуры Entity Framework Core никакой дальнейшей работы не осталось; следовательно, транзакция фиксируется, что приводит к применению изменений к БД и выдаче такого сообщения:

```
...
Committing transaction.
Фиксация транзакции.
...
```

До тех пор, пока транзакция не зафиксирована, изменения к БД не применяются и могут быть отброшены, если произойдет откат транзакции. Чтобы увидеть откат в действии, перейдите по ссылке http://localhost:5000/multi и измените значения в полях Salary на те, что представлены в табл. 24.5.

Таблица 24.5. Значения в полях Salary для неудачного выполнения транзакции

First Name (**Имя**)	Family Name (**Фамилия**)	Salary (**Оклад**)
Bob	Smith	900000000
Alice	Jones	300000

Значение в поле Salary для объекта Bob Smith не умещается в тип данных, сконфигурированный в модели данных, и после щелчка на кнопке Save All возникает исключение, которое прерывает обновление, из-за чего транзакция не зафиксируется. Поскольку оба обновления в табл. 24.5 выполняются внутри той же самой транзакции, ни одно из них к БД не применяется.

Выполнение независимых изменений

Потенциальным недостатком стандартного поведения транзакций является невозможность применения к БД успешных обновлений, если они сгруппированы с обновлением, которое терпит неудачу. Например, в предыдущем разделе допустимое обновление для объекта `Alice Jones` не прошло, т.к. оно было сгруппировано с неудачным обновлением для объекта `Bob Smith`. Если вы хотите изолировать каждое обновление, чтобы оно не подвергалось влиянию со стороны других неудавшихся обновлений, тогда можете вызывать метод `SaveChanges()` для каждого обновления (листинг 24.5).

Листинг 24.5. Выполнение независимых обновлений в файле `MultiController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      return View("EditAll", context.Employees);
    }

    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      foreach (Employee e in employees) {
        try {
          context.Update(e);
          context.SaveChanges();
        } catch (Exception) {
          context.Entry(e).State = EntityState.Detached;
        }
      }
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Каждый объект `Employee` передается методу `Update()` объекта контекста по отдельности, после чего вызывается метод `SaveChanges()`.

Использование блока `try...catch` гарантирует, что исключения, сгенерированные каким-то обновлением, не нарушат последующие обновления.

В результате для каждого обновления будет применяться отдельная транзакция, в чем можно удостовериться, запустив приложение с помощью `dotnet run`, перейдя по ссылке `http://localhost:5000/multi` и внеся изменения согласно табл. 24.5. Обновление для объекта `Bob Smith` по-прежнему терпит неудачу, но обновление для `Alice Jones` на этот раз применяется к БД (рис. 24.2).

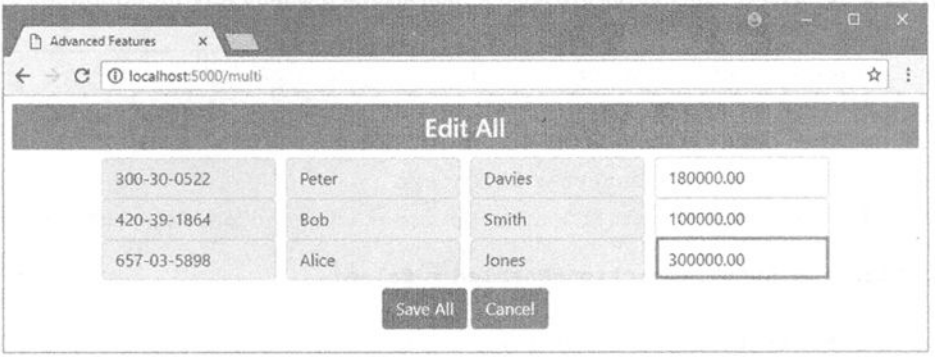

Рис. 24.2. Применение обновлений в собственных транзакциях

Совет. Обратите внимание, что состояние отслеживания объектов `Employee`, обновления которых потерпели неудачу, изменяется на `Detached` в конструкции `catch` блока `try...catch`. Без такого изменения инфраструктура Entity Framework Core включит неудавшееся обновление в последующий вызов метода `SaveChanges()`, что станет причиной еще одной ошибки и воспрепятствует применению другого обновления.

Просмотрев сгенерированные приложением журнальные сообщения, вы увидите, что для каждого объекта создается и фиксируется отдельная транзакция. Исключение, сгенерированное при обновлении объекта `Bob Smith`, предотвращает фиксацию транзакции для этого изменения, но не влияет на применение к БД остальных изменений.

Отключение автоматических транзакций

Если вы не хотите, чтобы инфраструктура Entity Framework Core автоматически использовала транзакции, тогда можете отключить их, изменив конфигурацию объекта контекста. В объектах контекста определено свойство `Database`, которое возвращает объект `DatabaseFacade`, обеспечивающий доступ к средству транзакций. В классе `DatabaseFacade` определено свойство, описанное в табл. 24.6, которое применяется для управления средством автоматических транзакций.

Таблица 24.6. Свойство `DatabaseFacade` для управления средством автоматических транзакций

Имя	Описание
`AutoTransactionsEnabled`	Установка этого свойства в `false` приведет к отключению средства автоматических транзакций

Воспользуйтесь свойством AutoTransactionsEnabled, чтобы отключить средство автоматических транзакций для класса AdvancedContext (листинг 24.6). Оно должно быть установлено для каждого создаваемого объекта, т.е. применение конструктора будет гарантировать согласованные результаты в приложении ASP.NET Core MVC, где используется внедрение зависимостей.

Листинг 24.6. Отключение автоматических транзакций в файле AdvancedContext.cs из папки Models

```
using Microsoft.EntityFrameworkCore;
using System;

namespace AdvancedApp.Models {

  public class AdvancedContext : DbContext {

    public AdvancedContext(DbContextOptions<AdvancedContext> options)
        : base(options) {
      Database.AutoTransactionsEnabled = false;
    }

    public DbSet<Employee> Employees { get; set; }

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

Восстановите исходную реализацию метода UpdateAll(), чтобы все обновления производились с помощью единственного вызова метода SaveChanges(), как показано в листинге 24.7. Ранее это означало бы, что все обновления были выполнены в одной транзакции.

Листинг 24.7. Выполнение обновлений в файле MultiController.cs из папки Controllers

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using Microsoft.EntityFrameworkCore;

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      return View("EditAll", context.Employees);
    }
```

```
    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      context.UpdateRange(employees);
      context.SaveChanges();
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Чтобы посмотреть, каким образом внесенная в конфигурацию модификация изменила вывод, запустите приложение посредством `dotnet run` и перейдите по ссылке `http://localhost:5000/multi`. Внесите изменения из табл. 24.7 и щелкните на кнопке Save Changes (Сохранить изменения).

Таблица 24.7. Измененные значения в полях Salary **для тестирования отключения автоматических транзакций**

First Name **(Имя)**	Family Name **(Фамилия)**	Salary **(Оклад)**
Peter	Davies	200000
Bob	Smith	900000000
Alice	Jones	200000

После щелчка на кнопке Save All вы получите исключение из-за того, что для объекта `Bob Smith` было указано значение, которое слишком велико, чтобы уместиться в тип данных столбца `Salary` в таблице БД. Но поскольку нет транзакции, подлежащей откату, другие изменения применяются к БД (рис. 24.3).

Рис. 24.3. Работа без автоматических транзакций

Совет. При работе без транзакций не всегда получается тот же самый результат, т.к. сервер баз данных или пакет поставщика могут реагировать по-разному. Важно помнить о том, что нельзя рассчитывать на откат всех обновлений, если одно из них терпит неудачу.

Использование явных транзакций

Стандартного поведения достаточно в большинстве проектов, но вы можете заставить инфраструктуру Entity Framework Core явно применять транзакцию для группирования вместе разрозненных операций. Поступать так полезно, если необходимо изменить способ настройки транзакций или выполнять дополнительные задачи перед принятием решения о том, фиксировать или производить откат набора обновлений.

Видоизмените метод `UpdateAll()` контроллера `Multi`, чтобы он использовал транзакцию явно, как показано в листинге 24.8.

Листинг 24.8. Применение транзакции в файле `MultiController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      return View("EditAll", context.Employees);
    }

    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      context.Database.BeginTransaction();
      try {
        context.UpdateRange(employees);
        context.SaveChanges();
        context.Database.CommitTransaction();
      } catch (Exception) {
        context.Database.RollbackTransaction();
      }
      return RedirectToAction(nameof(Index));
    }
  }
}
```

Прямой доступ к транзакциям осуществляется через свойство `Database` объекта контекста, возвращающее объект `DatabaseFacade`, который определяет относящиеся к транзакциям члены, описанные в табл. 24.8.

Таблица 24.8. Члены DatabaseFacade, связанные с транзакциями

Имя	Описание
`BeginTransaction()`	Этот метод создает новую транзакцию. Существует его асинхронная версия по имени `BeginTransactionAsync()`
`CommitTransaction()`	Этот метод фиксирует текущую транзакцию
`RollbackTransaction()`	Этот метод выполняет откат текущей транзакции
`CurrentTransaction`	Это свойство возвращает текущую транзакцию

В листинге 24.8 вызывается метод `BeginTransaction()` для запуска новой транзакции и затем выполняются обновления. После вызова метода `SaveChanges()` инфраструктура Entity Framework Core отправляет серверу баз данных SQL-команды для выполнения обновлений, но не фиксирует транзакцию автоматически. Взамен изменения не применяются к БД до тех пор, пока не будет вызван метод `CommitTransaction()`. Блок `try...catch` используется для обработки любых ошибок, о которых сообщает сервер баз данных, путем вызова метода `RollbackTransaction()`, отменяющего изменения.

Понятие освобождения транзакций

Вы будете часто видеть транзакции, создаваемые с помощью конструкции `using`, которой передается результат из метода `BeginTransaction()`, например:

```
...
using (var transaction = context.Database.BeginTransaction()) {
  // ...выполнение операций...
  context.Database.CommitTransaction();
}
...
```

При освобождении транзакция автоматически откатывается, а идея применения конструкции `using` заключается в том, что она гарантирует освобождение транзакции, когда она больше не требуется, предотвращая накапливание транзакций в ожидании отката и уничтожение их объектов.

В приложении ASP.NET Core MVC вы не обязаны беспокоиться об освобождении транзакций, потому что объекты освобождаются, когда завершается метод действия. Это означает, что после выполнения метода действия произойдет откат транзакции, если только не был явно вызван метод `CommitTransaction()` или `RollbackTransaction()`.

Включение в транзакцию других операций

В предыдущем примере было показано, как работать с транзакциями напрямую, но не выполнялись какие-либо задачи, которые работали бы отличающимся от автоматических транзакций образом. Распространенной причиной использования транзакций напрямую является выполнение дополнительной работы после отправки обновлений в БД перед определением, должны они быть зафиксированы или требуется произвести откат. В качестве демонстрации добавьте в метод `UpdateAll()` проверку, которая запрашивает БД с целью установки ограничения на совокупное значение свойств `Salary`, и осуществляет откат транзакции, если общая сумма слишком велика (листинг 24.9).

Листинг 24.9. Выполнение дополнительной работы в файле `MultiController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using System.Linq;

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      return View("EditAll", context.Employees);
    }

    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      context.Database.BeginTransaction();
      context.UpdateRange(employees);
      context.SaveChanges();
      if (context.Employees.Sum(e => e.Salary) < 1_000_000) {
        context.Database.CommitTransaction();
      } else {
        context.Database.RollbackTransaction();
        throw new Exception("Salary total exceeds limit");
      }
      return RedirectToAction(nameof(Index));
    }
  }
}
```

После вызова метода `SaveChanges()` посредством метода `Sum()` получается общая сумма значений `Salary`. Транзакция фиксируется, если общая сумма меньше 1 000 000, а иначе она подвергается откату с генерацией исключения. Чтобы увидеть эффект, запустите приложение, перейдите по ссылке `http://localhost:5000/multi` и измените значения в полях Salary. Щелкните на кнопке Save All, в результате чего либо изменения применяются, либо генерируется исключение с отбрасыванием изменений (рис. 24.4).

Такая дополнительная проверка невозможна, когда используется средство автоматических транзакций, потому что транзакция зафиксируется, как только будет вызван метод `SaveChanges()`. Работая напрямую с транзакцией, перед принятием решения относительно результата можно решать дополнительные задачи.

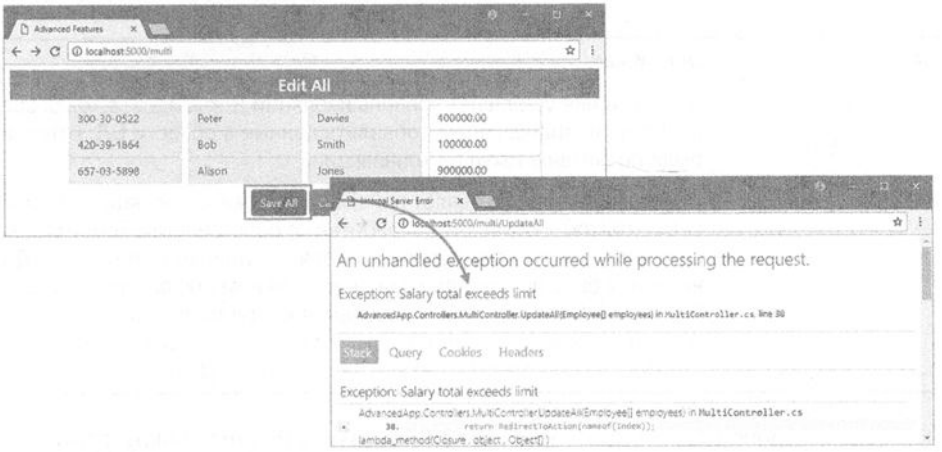

Рис. 24.4. Выполнение дополнительной работы в транзакции

Изменение уровня изоляции транзакций

Уровень изоляции транзакций указывает, каким образом сервер баз данных обрабатывает обновления, которые пока еще не были зафиксированы. В некоторых приложениях может быть важно изолировать изменения, вносимые одним клиентом, от остальных клиентов до тех пор, пока они не зафиксируются. В других приложениях изоляция такого вида менее важна, и за счет сокращения степени, в которой клиенты защищены от незафиксированных изменений, вносимых другими, можно улучшить производительность. Уровень изоляции для транзакции указывается с помощью перечисления `IsolationLevel`, значения которого описаны в табл. 24.9.

Таблица 24.9. Значения `IsolationLevel`

Имя	Описание
`Chaos`	Это значение определено нечетко, но когда оно поддерживается, то обычно ведет себя так, как будто транзакции отключены. Изменения, вносимые в БД, видны остальным клиентам до их фиксации и зачастую поддержка отката отсутствует. В SQL Server такой уровень изоляции не поддерживается
`ReadUncomitted`	Это значение представляет самый низкий уровень изоляции, который обычно поддерживается. Транзакции с таким уровнем изоляции могут читать изменения, сделанные другими транзакциями, которые не были зафиксированы
`ReadComitted`	Это значение является стандартным уровнем изоляции, который применяется, если никакое значение не указано. Другие транзакции по-прежнему способны вставлять или удалять данные между обновлениями, вносимыми текущей транзакцией, что может привести к несогласованным результатам запросов
`RepeatableRead`	Это значение представляет более высокий уровень изоляции, который запрещает другим транзакциям модификацию данных, прочитанных текущей транзакцией, что обеспечивает согласованные результаты запросов

Окончание табл. 24.9

Имя	Описание
`Serializable`	Это значение усиливает уровень изоляции `RepeatableRead`, запрещая другим транзакциям добавлять данные в области БД, которые были прочитаны текущей транзакцией
`Snapshot`	Это значение представляет наивысший уровень изоляции и гарантирует, что каждая транзакция работает с собственным моментальным снимком данных. Такой уровень изоляции требует изменения БД и не может быть произведен с использованием инфраструктуры Entity Framework Core. Подробные сведения доступны по ссылке `https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server`

Чем выше уровень изоляции, тем меньше взаимодействие между транзакциями, но больше блокирование, которое вынужден применять сервер баз данных, что оказывает влияние на производительность, если несколько клиентов одновременно пытаются модифицировать те же самые данные. При выборе уровня изоляции необходимо достичь компромисса между производительностью и несогласованностью данных. Для многих приложений, особенно в проектах ASP.NET Core MVC, практически не возникает потребность в изменении уровня изоляции, потому что стандартное значение обеспечит приемлемый компромисс. Однако для некоторых приложений (в особенности работающих с чрезвычайно сложными БД) риск взаимодействия между транзакциями может оказаться проблемой.

Избегание ловушки, связанной с высокими уровнями изоляции транзакций

У вас может возникнуть соблазн выбирать высокий уровень изоляции транзакций, просто чтобы пребывать в безопасности. Безусловно, это избавит от трех проблем, описанных далее в главе, но может также замедлить работу приложения. Серверы баз данных усиливают изоляцию транзакций за счет блокирования доступа к частям БД. Чем выше уровень изоляции, тем больше блокировок требуется и более вероятно, что обновления начнут ставиться в очередь, заставляя клиентов ожидать завершения запущенных ранее транзакций. Неправильный выбор уровня изоляции не даст никаких преимуществ приложению, но может привести к снижению производительности.

Грязное чтение, фантомные строки и невоспроизводимое чтение

Когда транзакции мешают друг другу, могут возникнуть три потенциальные проблемы. Принимая решение относительно уровня изоляции своей транзакции, вы на самом деле сообщаете серверу баз данных о том, какую из этих потенциальных проблем вы готовы принять, и какую долю производительности принести в жертву, чтобы избежать их. Внесите в контроллер `Multi` изменения, как демонстрируется в листинге 24.10.

Листинг 24.10. Использование уровней изоляции в файле `MultiController.cs` из папки `Controllers`

```
using AdvancedApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
```

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

namespace AdvancedApp.Controllers {

  public class MultiController : Controller {
    private AdvancedContext context;
    private ILogger<MultiController> logger;
    private IsolationLevel level = IsolationLevel.ReadUncommitted;

    public MultiController(AdvancedContext ctx, ILogger<MultiController> log)
    {
      context = ctx;
      logger = log;
    }

    public IActionResult Index() {
      context.Database.BeginTransaction(level);
      return View("EditAll", context.Employees);
    }

    [HttpPost]
    public IActionResult UpdateAll(Employee[] employees) {
      context.Database.BeginTransaction(level);
      context.UpdateRange(employees);
      Employee temp = new Employee {
        SSN = "00-00-0000",
        FirstName = "Temporary",
        FamilyName = "Row",
        Salary = 0
      };
      context.Add(temp);
      context.SaveChanges();
      System.Threading.Thread.Sleep(5000);
      context.Remove(temp);
      context.SaveChanges();
      if (context.Employees.Sum(e => e.Salary) < 1_000_000) {
        context.Database.CommitTransaction();
      } else {
        context.Database.RollbackTransaction();
        logger.LogError("Salary total exceeds limit");
      }
      return RedirectToAction(nameof(Index));
    }

    public string ReadTest() {
      decimal firstSum = context.Employees.Sum(e => e.Salary);
      System.Threading.Thread.Sleep(5000);
      decimal secondSum = context.Employees.Sum(e => e.Salary);
      return $"Repeatable read results - first: {firstSum}, "
        + $"second: {secondSum}";
    }
  }
}
```

Уровень изоляции устанавливается как аргумент метода `BeginTransaction()`, который является расширяющим методом, определенным в пространстве имен `Microsoft.EntityFrameworkCore`.

В листинге 24.10 выбран уровень `ReadUncomitted` для транзакции, которая осуществляет запрос БД в методе действия `Index()` и обновляет БД в методе действия `UpdateAll()`. В методе `UpdateAll()` вносятся обновления, полученные от пользователя, в БД добавляется временный объект `Employee` и затем он удаляется. Транзакция фиксируется, только если сумма значений `Salary` оказывается меньше 1 000 000, а иначе производится откат. Чтобы замедлить метод `UpdateAll()` и облегчить выявление эффектов от взаимодействия транзакций, был добавлен вызов метода `Thread.Sleep()`, который задерживает фиксацию или откат транзакции на 5 секунд.

Две из потенциальных проблем можно увидеть, запустив приложение посредством `dotnet run`, открыв два окна браузера и перейдя в обоих окнах по ссылке `http://localhost:5000/multi`. Измените значение в поле Salary для объекта `Alice Jones` на `900000` в первом окне браузера и щелкните на кнопке Save All. Перезагрузите страницу во втором окне браузера до того, как истечет пятисекундный период, и вы получите результаты, показанные на рис. 24.5.

Рис. 24.5. Изменения, сделанные другой транзакцией

Внимание! У вас не всегда будет возможность воспроизвести эти проблемы, особенно в реальных проектах, где контролировать условия гораздо труднее. Вы должны выбирать уровень изоляции, основываясь на том уровне, который необходим в принципе, даже если не удается воспроизвести специфическую проблему во время тестирования.

Несмотря на то что впоследствии происходит откат изменений, существует промежуток, в течение которого уровень изоляции `ReadUncomitted` позволяет читать изменения в другой транзакции. Временная строка является примером фантомной строки, а отображаемое для объекта `Alice Jones` значение `Salary` — примером грязного чтения.

Чтобы увидеть третью проблему, перейдите по ссылке `http://localhost:5000/multi/readtest` в одном из окон браузера. Запрос нацелен на метод действия `ReadTest()`, добавленный в листинге 24.10, который выполняет тот же самый запрос к БД до и после пятисекундной паузы.

Метод `ReadTest()` возвращает строку, поэтому добавлять представление в проект не нужно, и производит результат, который содержит одно и то же значение для двух чтений:

```
Repeatable read results - first: 500000.00, second: 500000.00
```

Перейдите по ссылке `http://localhost:5000/multi` во втором окне браузера, измените значение в поле Salary для объекта `Alice Jones` на `900000` и щелкните на кнопке Save All. До истечения пятисекундной паузы перезагрузите страницу в первом окне браузера, чтобы повторить два запроса. На этот раз вы увидите, что каждый запрос получает отличающийся результат:

```
Repeatable read results - first: 1200000.00, second: 500000.00
```

Возникла проблема невоспроизводимого чтения, когда два запроса, выполняемые той же самой транзакцией, возвращают разные результаты. Если вы измените уровень изоляции для транзакции в контроллере, то не столкнетесь с указанными проблемами.

Резюме

В главе был описан способ поддержки транзакций инфраструктурой Entity Framework Core, начиная со средства автоматических транзакций и заканчивая работой с транзакциями напрямую. Было показано, каким образом отключать транзакции, как выполнять дополнительную работу в виде части транзакции и каким образом сообщать инфраструктуре Entity Framework Core о том, когда фиксировать, а когда откатывать изменения в БД. Глава завершилась объяснением различных уровней изоляции, которые можно применять с транзакциями, и связанных с ними компромиссов в плане производительности.

Вот и все, что я должен был рассказать об использовании инфраструктуры Entity Framework Core при разработке приложений ASP.NET Core MVC. Я начал с представления основ работы с БД, объяснил распространенные проблемы, с которыми вы можете столкнуться, и описал все важные функциональные средства Entity Framework Core. Желаю вам успехов в разработке собственных проектов ASP.NET Core MVC и Entity Framework Core MVC и надеюсь на то, что вы получили не меньшее удовольствие от чтения этой книги, чем я во время ее написания.

Предметный указатель

В

Д

З

И

K

M

О

П

Р

С

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Entity Framework Core 2 для ASP.NET Core MVC для профессионалов

Эффективно моделируйте, отображайте и получайте доступ к данным с помощью Entity Framework Core 2 — новейшего выпуска инфраструктуры объектно-реляционного отображения от Microsoft. Вы получите возможность обращаться к данным с использованием объектов .NET через самый распространенный уровень доступа к данным, применяемый в проектах ASP.NET Core MVC 2.

Автор многочисленных бестселлеров Адам Фримен объясняет, как извлечь максимальную пользу из Entity Framework Core 2 в проектах MVC. Сначала он описывает различные способы моделирования данных посредством инфраструктуры Entity Framework Core 2 и разнообразные типы баз данных, которые могут применяться. Затем он показывает, каким образом использовать Entity Framework Core 2 в собственных проектах MVC, начиная с основных элементов и заканчивая наиболее сложными и развитыми функциональными возможностями, и в ходе изложения предоставляет вам все необходимые знания.

Благодаря этой книге, вы ...

- Обретете глубокое понимание архитектуры Entity Framework Core 2
- Научитесь создавать базы данных с применением модели данных MVC
- Узнаете, как создавать модели MVC с использованием существующей базы данных
- Обеспечите доступ к данным в приложении MVC с применением Entity Framework Core 2
- Научитесь использовать Entity Framework в веб-службах REST

Каждая тема раскрывается кратко и понятно с приведением всех деталей, необходимых для подлинно эффективного освоения. Наиболее важным средствам дается всеобъемлющее толкование, при этом в главах затрагиваются часто возникающие проблемы и предлагаются способы их предотвращения.

НА ВЕБ-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу: http://www.williamspublishing.com/Books/978-5-907114-86-9.html.

Категория: программирование
Предмет рассмотрения: Entity Framework Core 2
Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-907114-86-9

9 785907 114869 19005

ДИАЛЕКТИКА

www.williamspublishing.com

Apress®