Entity Framework Core в действии

Джон П. Смит

MANNING

ДМК издательство

DOT NET .RU

Джон П. Смит

Entity Framework Core в действии

Entity Framework Core in Action

SECOND EDITION

JON P. SMITH
Foreword by Julie Lerman

MANNING
Shelter Island

Entity Framework Core в действии

ДЖОН П. СМИТ
Предисловие Джули Лерман

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.4
ББК 32.372
С50

Под редакцией сообщества .NET разработчиков DotNet.Ru

Смит Дж. П.
С50 Entity Framework Core в действии / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2022. – 690 с.: ил.

ISBN 978-5-93700-114-6

Entity Framework радикально упрощает доступ к данным в приложениях .NET. Этот простой в использовании инструмент объектно-реляционного отображения (ORM) позволяет писать код базы данных на чистом C#. Он автоматически отображает классы в таблицы базы данных, разрешает запросы со стандартными командами LINQ и даже генерирует SQL-код за вас.

Данная книга научит вас писать код для беспрепятственного взаимодействия с базой данных при работе с приложениями .NET. Следуя соответствующим примерам из обширного опыта автора книги, вы быстро перейдете от основ к продвинутым методам. Помимо новейших функциональных возможностей EF, в книге рассматриваются вопросы производительности, безопасности, рефакторинга и модульного тестирования.

Издание предназначено разработчикам .NET, знакомым с реляционными базами данных.

УДК 004.4
ББК 32.372

ISBN 978-1-6172-9836-3 (англ.)
ISBN 978-5-93700-114-6 (рус.)

Оглавление

Часть I ■ НАЧАЛО 34
1 ■ Введение в Entity Framework Core 36
2 ■ Выполнение запроса к базе данных 63
3 ■ Изменение содержимого базы данных 102
4 ■ Использование EF Core в бизнес-логике 139
5 ■ Использование EF Core в веб-приложениях ASP.NET Core 175
6 ■ Советы и техники, касающиеся чтения и записи данных с EF Core 215

Часть II ■ ОБ ENTITY FRAMEWORK В ДЕТАЛЯХ 250
7 ■ Настройка нереляционных свойств 252
8 ■ Конфигурирование связей 291
9 ■ Управление миграциями базы данных 339
10 ■ Настройка расширенных функций и разрешение конфликтов параллельного доступа 382
11 ■ Углубляемся в DbContext 420

Часть III ■ ИСПОЛЬЗОВАНИЕ ENTITY FRAMEWORK CORE В РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ 462
12 ■ Использование событий сущности для решения проблем бизнес-логики 464
13 ■ Предметно-ориентированное проектирование и другие архитектурные подходы 492
14 ■ Настройка производительности в EF Core 530
15 ■ Мастер-класс по настройке производительности запросов к базе данных 561
16 ■ Cosmos DB, CQRS и другие типы баз данных 595
17 ■ Модульное тестирование приложений, использующих EF Core 634

Содержание

Предисловие 21
Введение 23
Благодарности 25
Об этой книге 26
Об авторе 32
Об изображении на обложке 33

Часть I НАЧАЛО 34

1 ***Введение в Entity Framework Core*** 36
1.1 Что вы узнаете из этой книги 37
1.2 Мой «момент озарения» 38
1.3 Несколько слов для разработчиков EF6.x 40
1.4 Обзор EF Core 40
1.4.1 Недостатки инструментов объектно-реляционного отображения 41
1.5 Что насчет нереляционных (NoSQL) баз данных? 42
1.6 Ваше первое приложение, использующее EF Core 42
1.6.1 Что нужно установить 43
1.6.2 Создание собственного консольного приложения .NET Core с помощью EF Core 44
1.7 База данных, к которой будет обращаться MyFirstEfCoreApp 45
1.8 Настройка приложения MyFirstEfCoreApp 47
1.8.1 Классы, которые отображаются в базу данных: Book и Author 47
1.8.2 DbContext 48
1.9 Заглянем под капот EF Core 49
1.9.1 Моделирование базы данных 50
1.9.2 Чтение данных 51
1.9.3 Обновление 54
1.10 Этапы разработки EF Core 57
1.11 Стоит ли использовать EF Core в своем следующем проекте? 58
1.11.1 .NET – это программная платформа будущего, и она будет быстрой! 58
1.11.2 Открытый исходный код и открытые сообщения 59
1.11.3 Мультиплатформенные приложения и разработка 59
1.11.4 Быстрая разработка и хорошие функциональные возможности 59
1.11.5 Хорошая поддержка 60
1.11.6 Всегда высокая производительность 60
1.12 Когда не следует использовать EF Core? 61
Резюме 61

2 *Выполнение запроса к базе данных* ... 63

2.1 Закладываем основу: наш сайт по продаже книг ... 64
2.1.1 Реляционная база данных приложения Book App ... 64
2.1.2 Другие типы связей, не описанные в этой главе ... 67
2.1.3 База данных – все таблицы ... 68
2.1.4 Классы, которые EF Core отображает в базу данных ... 70
2.2 Создание DbContext ... 72
2.2.1 Определение DbContext приложения: EfCoreContext ... 72
2.2.2 Создание экземпляра DbContext приложения ... 73
2.2.3 Создание базы данных для своего приложения ... 74
2.3 Разбираемся с запросами к базе данных ... 75
2.3.1 Доступ к свойству DbContext приложения ... 76
2.3.2 Серия команд LINQ / EF Core ... 76
2.3.3 Команда выполнения ... 76
2.3.4 Два типа запросов к базе данных ... 77
2.4 Загрузка связанных данных ... 78
2.4.1 Немедленная загрузка: загрузка связей с первичным классом сущности ... 78
2.1.2 Явная загрузка: загрузка связей после первичного класса сущности ... 81
2.4.3 Выборочная загрузка: загрузка определенных частей первичного класса сущности и любых связей ... 82
2.4.4 Отложенная загрузка: загрузка связанных данных по мере необходимости ... 83
2.5 Использование вычисления на стороне клиента: адаптация данных на последнем этапе запроса ... 85
2.6 Создание сложных запросов ... 88
2.7 Знакомство с архитектурой приложения Book App ... 92
2.8 Добавляем сортировку, фильтрацию и разбиение на страницы ... 93
2.8.1 Сортировка книг по цене, дате публикации и оценкам покупателей ... 94
2.8.2 Фильтрация книг по году публикации, категориям и оценкам покупателей ... 95
2.8.3 Другие параметры фильтрации: поиск текста по определенной строке ... 96
2.8.4 Разбиение книг на страницы в списке ... 98
2.9 Собираем все вместе: объединение объектов запроса ... 99
Резюме ... 100

3 *Изменение содержимого базы данных* ... 102

3.1 Представляем свойство сущности State ... 103
3.2 Создание новых строк в таблице ... 103
3.2.1 Самостоятельное создание отдельной сущности ... 104
3.2.2 Создание книги с отзывом ... 105
3.3 Обновление строк базы данных ... 109
3.3.1 Обработка отключенных обновлений в веб-приложении ... 111
3.4 Обработка связей в обновлениях ... 117
3.4.1 Основные и зависимые связи ... 118
3.4.2 Обновление связей «один к одному»: добавляем PriceOffer в книгу ... 119
3.4.3 Обновление связей «один ко многим»: добавляем отзыв в книгу ... 123
3.4.4 Обновление связи «многие ко многим» ... 127
3.4.5 Расширенная функция: обновление связей через внешние ключи ... 132
3.5 Удаление сущностей ... 133
3.5.1 Мягкое удаление: использование глобального фильтра запросов, чтобы скрыть сущности ... 133
3.5.2 Удаление только зависимой сущности без связей ... 135

3.5.3 Удаление основной сущности, у которой есть связи ... 135
3.5.4 Удаление книги с зависимыми связями ... 136
Резюме ... 137

4 Использование EF Core в бизнес-логике ... 139
4.1 Вопросы, которые нужно задать, и решения, которые нужно принять, прежде чем начать писать код ... 140
4.1.1 Три уровня сложности кода бизнес-логики ... 141
4.2 Пример сложной бизнес-логики: обработка заказа на приобретение книги ... 143
4.3 Использование паттерна проектирования для реализации сложной бизнес-логики ... 144
4.3.1 Пять правил по созданию бизнес-логики, использующей EF Core ... 144
4.4 Реализация бизнес-логики для обработки заказа ... 146
4.4.1 Правило 1: бизнес-логика требует определения структуры базы данных ... 147
4.4.2 Правило 2: ничто не должно отвлекать от бизнес-логики ... 148
4.4.3 Правило 3: бизнес-логика должна думать, что работает с данными в памяти ... 149
4.4.4 Правило 4: изолируйте код доступа к базе данных в отдельный проект ... 152
4.4.5 Правило 5: бизнес-логика не должна вызывать метод EF Core, SaveChanges ... 153
4.4.6 Собираем все вместе: вызов бизнес-логики для обработки заказов ... 156
4.4.7 Размещение заказа в приложении Book App ... 157
4.4.8 Плюсы и минусы паттерна сложной бизнес-логики ... 159
4.5 Пример простой бизнес-логики: ChangePriceOfferService ... 159
4.5.1 Мой подход к проектированию простой бизнес-логики ... 160
4.5.2 Пишем код класса ChangePriceOfferService ... 160
4.5.3 Плюсы и минусы этого паттерна бизнес-логики ... 161
4.6 Пример валидации: добавление отзыва в книгу с проверкой ... 162
4.6.1 Плюсы и минусы этого паттерна бизнес-логики ... 163
4.7 Добавление дополнительных функций в обработку вашей бизнес-логики ... 163
4.7.1 Валидация данных, которые вы записываете в базу ... 164
4.7.2 Использование транзакций для объединения кода бизнес-логики в одну логическую атомарную операцию ... 168
4.7.3 Использование класса RunnerTransact2WriteDb ... 172
Резюме ... 173

5 Использование EF Core в веб-приложениях ASP.NET Core ... 175
5.1 Знакомство с ASP.NET Core ... 176
5.2 Разбираемся с архитектурой приложения Book App ... 176
5.3 Внедрение зависимостей ... 177
5.3.1 Почему нужно знать, что такое внедрение зависимостей, работая с ASP.NET Core ... 179
5.3.2 Базовый пример внедрения зависимостей в ASP.NET Core ... 179
5.3.3 Жизненный цикл сервиса, созданного внедрением зависимостей ... 180
5.3.4 Особые соображения, касающиеся приложений Blazor Server ... 182
5.4 Делаем DbContext приложения доступным, используя внедрение зависимостей ... 182
5.4.1 Предоставление информации о расположении базы данных ... 183

5.4.2 Регистрация DbContext приложения у поставщика внедрения зависимостей ... 184
5.4.3 Регистрация фабрики DbContext у поставщика внедрения зависимостей ... 185
5.5 Вызов кода доступа к базе данных из ASP.NET Core ... 186
5.5.1 Краткое изложение того, как работает паттерн ASP.NET Core MVC, и термины, которые он использует ... 187
5.5.2 Где находится код EF Core в приложении Book App? ... 187
5.6 Реализация страницы запроса списка книг ... 189
5.6.1 Внедрение экземпляра DbContext приложения через внедрение зависимостей ... 189
5.6.2 Использование фабрики DbContext для создания экземпляра DbContext ... 191
5.7 Реализация методов базы данных как сервиса внедрения зависимостей ... 193
5.7.1 Регистрация класса в качестве сервиса во внедрении зависимостей ... 194
5.7.2 Внедрение ChangePubDateService в метод действия ASP.NET ... 195
5.7.3 Улучшаем регистрацию классов доступа к базе данных как сервисов ... 196
5.8 Развертывание приложения ASP.NET Core с базой данных ... 199
5.8.1 Местонахождение базы данных на веб-сервере ... 200
5.8.2 Создание и миграция базы данных ... 201
5.9 Использование функции миграции в EF Core для изменения структуры базы данных ... 201
5.9.1 Обновление рабочей базы данных ... 202
5.9.2 Заставляем приложение обновить базу данных при запуске ... 203
5.10 Использование async/await для лучшей масштабируемости ... 206
5.10.1 Чем паттерн async/await полезен в веб-приложении, использующем EF Core ... 207
5.10.2 Где использовать async/await для доступа к базе данных? ... 208
5.10.3 Переход на версии команд EF Core с async/await ... 208
5.11 Выполнение параллельных задач: как предоставить DbContext ... 210
5.11.1 Получение экземпляра DbContext для параллельного запуска ... 211
5.11.2 Запуск фоновой службы в ASP.NET Core ... 212
5.11.3 Другие способы получения нового экземпляра DbContext ... 213
Резюме ... 213

6 *Советы и техники, касающиеся чтения и записи данных с EF Core* ... 215

6.1 Чтение из базы данных ... 216
6.1.1 Этап ссылочной фиксации в запросе ... 216
6.1.2 Понимание того, что делает метод AsNoTracking и его разновидности ... 218
6.1.3 Эффективное чтение иерархических данных ... 220
6.1.4 Понимание того, как работает метод Include ... 222
6.1.5 Обеспечение отказоустойчивости загрузки навигационных коллекций ... 224
6.1.6 Использование глобальных фильтров запросов в реальных ситуациях ... 225
6.1.7 Команды LINQ, требующие особого внимания ... 230
6.1.8 Использование AutoMapper для автоматического построения запросов с методом Select ... 232
6.1.9 Оценка того, как EF Core создает класс сущности при чтении данных ... 235
6.2 Запись данных в базу с EF Core ... 240
6.2.1 Оценка того, как EF Core записывает сущности или связи

в базу данных ... 240
6.2.2 Оценка того, как DbContext обрабатывает запись сущностей и связей ... 242
6.2.3 Быстрый способ копирования данных со связями ... 246
6.2.4 Быстрый способ удалить сущность ... 247
Резюме ... 248

Часть II ОБ ENTITY FRAMEWORK В ДЕТАЛЯХ ... 250

7 Настройка нереляционных свойств ... 252
7.1 Три способа настройки EF Core ... 253
7.2 Рабочий пример настройки EF Core ... 254
7.3 Конфигурация по соглашению ... 257
7.3.1 Соглашения для классов сущностей ... 257
7.3.2 Соглашения для параметров в классе сущности ... 258
7.3.3 Условные обозначения для имени, типа и размера ... 258
7.3.4 По соглашению поддержка значения NULL для свойства основана на типе .NET ... 259
7.3.5 Соглашение об именах EF Core определяет первичные ключи ... 259
7.4 Настройка с помощью аннотаций данных ... 260
7.4.1 Использование аннотаций из пространства имен System.ComponentModel.DataAnnotations ... 261
7.4.2 Использование аннотаций из пространства имен System.ComponentModel.DataAnnotations.Schema ... 261
7.5 Настройка с использованием Fluent API ... 261
7.6 Исключение свойств и классов из базы данных ... 264
7.6.1 Исключение класса или свойства с помощью Data Annotations ... 264
7.6.2 Исключение класса или свойства с помощью Fluent API ... 265
7.7 Установка типа, размера и допустимости значений NULL для столбца базы данных ... 266
7.8 Преобразование значения: изменение данных при чтении из базы данных или записи в нее ... 267
7.9 Различные способы настройки первичного ключа ... 269
7.9.1 Настройка первичного ключа с помощью Data Annotations ... 269
7.9.2 Настройка первичного ключа через Fluent API ... 270
7.9.3 Настройка сущности как класса с доступом только на чтение ... 270
7.10 Добавление индексов в столбцы базы данных ... 271
7.11 Настройка именования на стороне базы данных ... 272
7.11.1 Настройка имен таблиц ... 273
7.11.2 Настройка имени схемы и группировки схем ... 273
7.11.3 Настройка имен столбцов базы данных в таблице ... 274
7.12 Настройка глобальных фильтров запросов ... 274
7.13 Применение методов Fluent API в зависимости от типа поставщика базы данных ... 275
7.14 Теневые свойства: сокрытие данных столбца внутри EF Core ... 276
7.14.1 Настройка теневых свойств ... 277
7.14.2 Доступ к теневым свойствам ... 277
7.15 Резервные поля: управление доступом к данным в классе сущности ... 278
7.15.1 Создание простого резервного поля, доступного через свойство чтения/записи ... 279
7.15.2 Создание столбца с доступом только на чтение ... 279
7.15.3 Сокрытие даты рождения внутри класса ... 280
7.15.4 Настройка резервных полей ... 281
7.16 Рекомендации по использованию конфигурации EF Core ... 283

7.16.1 Сначала используйте конфигурацию «По соглашению» ... 284
7.16.2 По возможности используйте Data Annotations ... 284
7.16.3 Используйте Fluent API для всего остального ... 284
7.16.4 Автоматизируйте добавление команд Fluent API по сигнатурам класса или свойства ... 285
Резюме ... 289

8 *Конфигурирование связей* ... 291
8.1 Определение терминов, относящихся к связям ... 292
8.2 Какие навигационные свойства нам нужны? ... 293
8.3 Настройка связей ... 294
8.4 Настройка связей по соглашению ... 295
8.4.1. Что делает класс классом сущности? ... 295
8.4.2 Пример класса сущности с навигационными свойствами ... 295
8.4.3 Как EF Core находит внешние ключи по соглашению ... 296
8.4.4 Поддержка значения null у внешних ключей: обязательные или необязательные зависимые связи ... 297
8.4.5 Внешние ключи: что произойдет, если не указать их? ... 298
8.4.6 Когда подход «По соглашению» не работает? ... 300
8.5 Настройка связей с помощью аннотаций данных ... 300
8.5.1 Аннотация ForeignKey ... 300
8.5.2 Аннотация InverseProperty ... 301
8.6 Команды Fluent API для настройки связей ... 302
8.6.1 Создание связи «один к одному» ... 303
8.6.2 Создание связи «один ко многим» ... 306
8.6.3 Создание связей «многие ко многим» ... 307
8.7 Управление обновлениями навигационных свойств коллекции ... 310
8.8 Дополнительные методы, доступные во Fluent API ... 312
8.8.1 OnDelete: изменение действия при удалении зависимой сущности ... 313
8.8.2 IsRequired: определение допустимости значения null для внешнего ключа ... 316
8.8.3 HasPrincipalKey: использование альтернативного уникального ключа ... 318
8.8.4 Менее используемые параметры в связях Fluent API ... 319
8.9 Альтернативные способы отображения сущностей в таблицы базы данных ... 320
8.9.1 Собственные типы: добавление обычного класса в класс сущности ... 320
8.9.2 Таблица на иерархию (TPH): размещение унаследованных классов в одной таблице ... 326
8.9.3 Таблица на тип (TPT): у каждого класса своя таблица ... 331
8.9.4 Разбиение таблицы: отображение нескольких классов сущностей в одну и ту же таблицу ... 333
8.9.5 Контейнер свойств: использование словаря в качестве класса сущности ... 335
Резюме ... 337

9 *Управление миграциями базы данных* ... 339
9.1 Как устроена эта глава ... 340
9.2 Сложности изменения базы данных приложения ... 340
9.2.1 Какие базы данных нуждаются в обновлении ... 341
9.2.2 Миграция, которая может привести к потере данных ... 342
9.3 Часть 1: знакомство с тремя подходами к созданию миграции ... 342
9.4 Создание миграции с помощью команды EF Core add migration ... 344
9.4.1 Требования перед запуском любой команды миграции EF Core ... 346

9.4.2 Запуск команды add migration ... 347
9.4.3 Заполнение базы данных с помощью миграции ... 348
9.4.4 Миграции и несколько разработчиков ... 349
9.4.5 Использование собственной таблицы миграций, позволяющей использовать несколько DbContext в одной базе данных ... 350
9.5 Редактирование миграции для обработки сложных ситуаций ... 353
9.5.1 Добавление и удаление методов MigrationBuilder внутри класса миграции ... 354
9.5.2 Добавление команд SQL в миграцию ... 355
9.5.3 Добавление собственных команд миграции ... 357
9.5.4 Изменение миграции для работы с несколькими типами баз данных ... 358
9.6 Использование сценариев SQL для создания миграций ... 360
9.6.1 Использование инструментов сравнения баз данных SQL для выполнения миграции ... 361
9.6.2 Написание кода сценариев изменения SQL для миграции базы данных вручную ... 363
9.6.3 Проверка соответствия сценариев изменения SQL модели базы данных EF Core ... 365
9.7 Использование инструмента обратного проектирования EF Core ... 366
9.7.1 Запуск команды обратного проектирования ... 367
9.7.2 Установка и запуск команды обратного проектирования Power Tools ... 368
9.7.3 Обновление классов сущности и DbContext при изменении базы данных ... 368
9.8 Часть 2: применение миграций к базе данных ... 369
9.8.1 Вызов метода Database.Migrate из основного приложения ... 370
9.8.2 Выполнение метода Database.Migrate из отдельного приложения ... 373
9.8.3 Применение миграции EF Core с помощью SQL-сценария ... 373
9.8.4 Применение сценариев изменения SQL с помощью инструмента миграций ... 375
9.9 Миграция базы данных во время работы приложения ... 375
9.9.1 Миграция, которая не содержит критических изменений ... 377
9.9.2 Работа с критическими изменениями, когда вы не можете остановить приложение ... 378
Резюме ... 380

10 *Настройка расширенных функций и разрешение конфликтов параллельного доступа* ... 382
10.1 DbFunction: использование пользовательских функций с EF Core ... 383
10.1.1 Настройка скалярной функции ... 384
10.1.2 Настройка табличной функции ... 386
10.1.3 Добавление кода пользовательской функции в базу данных ... 387
10.1.4 Использование зарегистрированной пользовательской функции в запросах к базе данных ... 388
10.2 Вычисляемый столбец: динамически вычисляемое значение столбца ... 389
10.3 Установка значения по умолчанию для столбца базы данных ... 392
10.3.1 Использование метода HasDefaultValue для добавления постоянного значения для столбца ... 394
10.3.2 Использование метода HasDefaultValueSql для добавления команды SQL для столбца ... 395
10.3.3 Использование метода HasValueGenerator для назначения генератора значений свойству ... 396
10.4 Последовательности: предоставление чисел в строгом порядке ... 397

10.5 Помечаем свойства, созданные базой данных ... 398
10.5.1 Помечаем столбец, создаваемый при добавлении или обновлении ... 399
10.5.2 Помечаем значение столбца как установленное при вставке новой строки ... 400
10.5.3 Помечаем столбец/свойство как «обычное» ... 401
10.6 Одновременные обновления: конфликты параллельного доступа ... 402
10.6.1 Почему конфликты параллельного доступа так важны? ... 403
10.6.2 Возможности решения конфликтов параллельного доступа в EF Core ... 404
10.6.3 Обработка исключения DbUpdateConcurrencyException ... 411
10.6.4 Проблема с отключенным параллельным обновлением ... 415
Резюме ... 419

11 *Углубляемся в DbContext* ... 420
11.1 Обзор свойств класса DbContext ... 421
11.2 Как EF Core отслеживает изменения ... 421
11.3 Обзор команд, которые изменяют свойство сущности State ... 423
11.3.1 Команда Add: вставка новой строки в базу данных ... 424
11.3.2 Метод Remove: удаление строки из базы данных ... 425
11.3.3 Изменение класса сущности путем изменения данных в нем ... 425
11.3.4 Изменение класса сущности путем вызова метода Update ... 426
11.3.5 Метод Attach: начать отслеживание существующего неотслеживаемого класса сущности ... 428
11.3.6 Установка свойства сущности State напрямую ... 428
11.3.7 TrackGraph: обработка отключенных обновлений со связями ... 429
11.4 Метод SaveChanges и как он использует метод ChangeTracker. DetectChanges ... 431
11.4.1 Как метод SaveChanges находит все изменения состояния ... 432
11.4.2 Что делать, если метод ChangeTracker.DetectChanges занимает слишком много времени ... 432
11.4.3 Использование состояния сущностей в методе SaveChanges ... 437
11.4.4 Перехват изменений свойства State с использованием события ... 441
11.4.5 Запуск событий при вызове методов SaveChanges и SaveChangesAsync ... 444
11.4.6 Перехватчики EF Core ... 445
11.5 Использование команд SQL в приложении EF Core ... 445
11.5.1 Методы FromSqlRaw/FromSqlInterpolated: использование SQL в запросе EF Core ... 447
11.5.2 Методы ExecuteSqlRaw и ExecuteSqlInterpolated: выполнение команды без получения результата ... 448
11.5.3 Метод Fluent API ToSqlQuery: отображение классов сущностей в запросы ... 448
11.5.4 Метод Reload: используется после команд ExecuteSql ... 450
11.5.5 GetDbConnection: выполнение собственных команд SQL ... 450
11.6 Доступ к информации о классах сущностей и таблицам базы данных ... 452
11.6.1 Использование context.Entry(entity).Metadata для сброса первичных ключей ... 452
11.6.2 Использование свойства context.Model для получения информации о базе данных ... 455
11.7 Динамическое изменение строки подключения DbContext ... 456
11.8 Решение проблем, связанных с подключением к базе данных ... 457
11.8.1 Обработка транзакций базы данных с использованием стратегии выполнения ... 458
11.8.2 Изменение или написание собственной стратегии исполнения ... 460
Резюме ... 460

Часть III ИСПОЛЬЗОВАНИЕ ENTITY FRAMEWORK CORE В РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ ... 462

12 *Использование событий сущности для решения проблем бизнес-логики* ... 464

12.1 Использование событий для решения проблем бизнес-логики ... 465
12.1.1 Пример использования событий предметной области ... 465
12.1.2 Пример событий интеграции ... 467
12.2 Определяем, где могут быть полезны события предметной области и интеграции ... 468
12.3 Где можно использовать события с EF Core? ... 468
12.3.1 Плюс: следует принципу разделения ответственностей ... 470
12.3.2 Плюс: делает обновления базы данных надежными ... 470
12.3.3 Минус: делает приложение более сложным ... 470
12.3.4 Минус: усложняет отслеживание потока исполнения кода ... 471
12.4 Реализация системы событий предметной области с EF Core ... 472
12.4.1 Создайте несколько классов событий предметной области, которые нужно будет вызвать ... 473
12.4.2 Добавьте код в классы сущностей, где будут храниться события предметной области ... 474
12.4.3 Измените класс сущности, чтобы обнаружить изменение, при котором вызывается событие ... 475
12.4.4 Создайте обработчики событий, соответствующие событиям предметной области ... 475
12.4.5 Создайте диспетчер событий, который находит и запускает правильный обработчик событий ... 477
12.4.6 Переопределите метод SaveChanges и вставьте вызов диспетчера событий перед вызовом этого метода ... 479
12.4.7 Зарегистрируйте диспетчер событий и все обработчики событий ... 480
12.5 Внедрение системы событий интеграции с EF Core ... 482
12.5.1 Создание сервиса, который обменивается данными со складом ... 484
12.5.2 Переопределение метода SaveChanges для обработки события интеграции ... 485
12.6 Улучшение события предметной области и реализаций событий интеграции ... 486
12.6.1 Обобщение событий: запуск до, во время и после вызова метода SaveChanges ... 487
12.6.2 Добавление поддержки асинхронных обработчиков событий ... 488
12.6.3 Несколько обработчиков событий для одного и того же события ... 489
12.6.4 Последовательности событий, в которых одно событие запускает другое ... 490
Резюме ... 491

13 *Предметно-ориентированное проектирование и другие архитектурные подходы* ... 492

13.1 Хорошая программная архитектура упрощает создание и сопровождение приложения ... 493
13.2 Развивающаяся архитектура приложения Book App ... 494
13.2.1 Создание модульного монолита для обеспечения реализации принципов разделения ответственностей ... 495
13.2.2 Использование принципов предметно-ориентированного проектирования в архитектуре и в классах сущностей ... 497

13.2.3 Применение чистой архитектуры согласно описанию Роберта Мартина ... 498
13.3 Введение в предметно-ориентированное проектирование на уровне класса сущности ... 498
13.4 Изменение сущностей приложения Book App, чтобы следовать предметно-ориентированному проектированию ... 499
13.4.1 Изменение свойств сущности Book на доступ только для чтения ... 500
13.4.2 Обновление свойств сущности Book с помощью методов в классе сущности ... 502
13.4.3 Управление процессом создания сущности Book ... 503
13.4.4 Разбор различий между сущностями и объектом-значением ... 505
13.4.5 Минимизация связей между классами сущностей ... 505
13.4.6 Группировка классов сущностей ... 506
13.4.7 Принимаем решение, когда бизнес-логику не следует помещать внутрь сущности ... 508
13.4.8 Применение паттерна «Ограниченный контекст» к DbContext приложения ... 510
13.5 Использование классов сущностей в стиле DDD в вашем приложении ... 511
13.5.1 Вызов метода доступа AddPromotion с помощью паттерна «Репозиторий» ... 512
13.5.2 Вызов метода доступа AddPromotion с помощью библиотеки GenericServices ... 515
13.5.3 Добавление отзыва в класс сущности Book через паттерн «Репозиторий» ... 517
13.5.4 Добавление отзыва в класс сущности Book с помощью библиотеки GenericServices ... 518
13.6 Обратная сторона сущностей DDD: слишком много методов доступа ... 519
13.7 Решение проблем с производительностью в DDD-сущностях ... 520
13.7.1 Добавить код базы данных в свои классы сущностей ... 521
13.7.2 Сделать конструктор Review открытым и написать код для добавления отзыва вне сущности ... 523
13.7.3 Использовать события предметной области, чтобы попросить обработчик событий добавить отзыв в базу данных ... 523
13.8 Три архитектурных подхода: сработали ли они? ... 524
13.8.1 Модульный монолит, реализующий принцип разделения ответственностей с помощью проектов ... 524
13.8.2 Принципы DDD как в архитектуре, так и в классах сущностей ... 526
13.8.3 Чистая архитектура согласно описанию Роберта С. Мартина ... 527
Резюме ... 528

14 *Настройка производительности в EF Core* ... 530
14.1 Часть 1: решаем, какие проблемы с производительностью нужно исправлять ... 531
14.1.1 Фраза «Не занимайтесь настройкой производительности на ранних этапах» не означает, что нужно перестать думать об этом ... 531
14.1.2 Как определить, что работает медленно и требует настройки производительности? ... 532
14.1.3 Затраты на поиск и устранение проблем с производительностью ... 534
14.2 Часть 2: методы диагностики проблем с производительностью ... 535
14.2.1 Этап 1. Получить хорошее общее представление, оценив опыт пользователей ... 536
14.2.2 Этап 2. Найти весь код доступа к базе данных, связанный с оптимизируемой функцией ... 537
14.2.3 Этап 3. Проверить SQL-код, чтобы выявить низкую производительность ... 538

14.3 Часть 3: методы устранения проблем с производительностью 540
14.4 Использование хороших паттернов позволяет приложению хорошо работать 541
14.4.1 Использование метода Select для загрузки только нужных столбцов 542
14.4.2 Использование разбиения по страницам и/или фильтрации результатов поиска для уменьшения количества загружаемых строк 542
14.4.3 Понимание того, что отложенная загрузка влияет на производительность базы данных 543
14.4.4 Добавление метода AsNoTracking к запросам с доступом только на чтение 543
14.4.5 Использование асинхронной версии команд EF Core для улучшения масштабируемости 544
14.4.6 Поддержание кода доступа к базе данных изолированным/слабосвязанным 544
14.5 Антипаттерны производительности: запросы к базе данных 545
14.5.1 Антипаттерн: отсутствие минимизации количества обращений к базе данных 545
14.5.2 Антипаттерн: отсутствие индексов для свойства, по которому вы хотите выполнить поиск 547
14.5.3 Антипаттерн: использование не самого быстрого способа загрузки отдельной сущности 547
14.5.4 Антипаттерн: перенос слишком большой части запроса данных на сторону приложения 548
14.5.5 Антипаттерн: вычисления вне базы данных 549
14.5.6 Антипаттерн: использование неоптимального SQL-кода в LINQ-запросе 550
14.5.7 Антипаттерн: отсутствие предварительной компиляции часто используемых запросов 550
14.6 Антипаттерны производительности: операции записи 552
14.6.1 Антипаттерн: неоднократный вызов метода SaveChanges 552
14.6.2 Антипаттерн: слишком большая нагрузка на метод DetectChanges 553
14.6.3 Антипаттерн: HashSet<T> не используется для навигационных свойств коллекции 554
14.6.4 Антипаттерн: использование метода Update, когда нужно изменить только часть сущности 555
14.6.5 Антипаттерн: проблема при запуске – использование одного большого DbContext 555
14.7 Паттерны производительности: масштабируемость доступа к базе данных 556
14.7.1 Использование пулов для снижения затрат на создание нового DbContext приложения 557
14.7.2 Добавление масштабируемости с незначительным влиянием на общую скорость 557
14.7.3 Повышение масштабируемости базы данных за счет упрощения запросов 558
14.7.4 Вертикальное масштабирование сервера базы данных 558
14.7.5 Выбор правильной архитектуры для приложений, которым требуется высокая масштабируемость 559
Резюме 559

15 *Мастер-класс по настройке производительности запросов к базе данных* 561
15.1 Настройка тестового окружения и краткое изложение четырех подходов к повышению производительности 562

15.2 Хороший LINQ: использование выборочного запроса ... 565
15.3 LINQ + пользовательские функции: добавляем SQL в код LINQ ... 568
15.4 SQL + Dapper: написание собственного SQL-кода ... 570
15.5 LINQ + кеширование: предварительное вычисление частей запроса, которое занимает много времени ... 573
15.5.1 Добавляем способ обнаружения изменений, влияющих на кешированные значения ... 574
15.5.2 Добавление кода для обновления кешированных значений ... 577
15.5.3 Добавление свойств в сущность Book с обработкой параллельного доступа ... 581
15.5.4 Добавление системы проверки и восстановления в систему событий ... 587
15.6 Сравнение четырех подходов к производительности с усилиями по разработке ... 589
15.7 Повышение масштабируемости базы данных ... 591
Резюме ... 593

16 *Cosmos DB, CQRS и другие типы баз данных* ... 595
16.1 Различия между реляционными и нереляционными базами данных ... 596
16.2 Cosmos DB и ее провайдер для EF Core ... 597
16.3 Создание системы CQRS с использованием Cosmos DB ... 598
16.4 Проектирование приложения с архитектурой CQRS с двумя базами данных ... 601
16.4.1 Создание события, вызываемого при изменении сущности Book ... 602
16.4.2 Добавление событий в метод сущности Book ... 603
16.4.3 Использование библиотеки EfCore.GenericEventRunner для переопределения BookDbContext ... 605
16.4.4 Создание классов сущностей Cosmos и DbContext ... 605
16.4.5 Создание обработчиков событий Cosmos ... 607
16.5 Структура и данные учетной записи Cosmos DB ... 610
16.5.1 Структура Cosmos DB с точки зрения EF Core ... 610
16.5.2 Как CosmosClass хранится в Cosmos DB ... 611
16.6 Отображение книг через Cosmos DB ... 613
16.6.1 Отличия Cosmos DB от реляционных баз данных ... 614
16.6.2 Основное различие между Cosmos DB и EF Core: миграция базы данных Cosmos ... 617
16.6.3 Ограничения поставщика базы данных EF Core 5 для Cosmos DB ... 618
16.7 Стоило ли использование Cosmos DB затраченных усилий? Да! ... 621
16.7.1 Оценка производительности системы CQRS с двумя базами данных в приложении Book App ... 622
16.7.2 Исправление функций, с которыми поставщик баз данных EF Core 5 для Cosmos DB не справился ... 626
16.7.3 Насколько сложно было бы использовать эту систему CQRS с двумя базами данных в своем приложении? ... 629
16.8 Отличия в других типах баз данных ... 630
Резюме ... 632

17 *Модульное тестирование приложений, использующих EF Core* ... 634
17.1 Знакомство с настройкой модульного теста ... 637
17.1.1 Окружение тестирования: библиотека модульного тестирования xUnit ... 638
17.1.2 Созданная мной библиотека для модульного тестирования приложений, использующих EF Core ... 639

17.2 Подготовка DbContext приложения к модульному тестированию 640
17.2.1 Параметры DbContext приложения передаются в конструктор 640
17.2.2 Настройка параметров DbContext приложения через OnConfiguring ... 641
17.3 Три способа смоделировать базу данных при тестировании приложений EF Core 643
17.4 Выбор между базой данных того же типа, что и рабочая, и базой данных SQLite in-memory 644
17.5 Использование базы данных промышленного типа в модульных тестах 647
17.5.1 Настройка строки подключения к базе данных, которая будет использоваться для модульного теста 647
17.5.2 Создание базы данных для каждого тестового класса для параллельного запуска тестов в xUnit 649
17.5.3 Убеждаемся, что схема базы данных актуальна, а база данных пуста 651
17.5.4 Имитация настройки базы данных, которую обеспечит миграция EF Core 655
17.6 Использование базы данных SQLite in-memory для модульного тестирования 656
17.7 Создание заглушки или имитации базы данных EF Core 659
17.8 Модульное тестирование базы данных Cosmos DB 662
17.9 Заполнение базы данных тестовыми данными для правильного тестирования кода 664
17.10 Решение проблемы, когда один запрос к базе данных нарушает другой этап теста 665
17.10.1 Код теста с методом ChangeTracker.Clear в отключенном состоянии 667
17.10.2 Код теста с несколькими экземплярами DbContext в отключенном состоянии 668
17.11 Перехват команд, отправляемых в базу данных 669
17.11.1 Использование расширения параметра LogTo для фильтрации и перехвата сообщений журналов EF Core 669
17.11.2 Использование метода ToQueryString для отображения сгенерированного SQL-кода из LINQ-запроса 672
Резюме 673

Приложение А. Краткое введение в LINQ 675
Предметный указатель 686

DOT
NET
.RU

Вступительное слово от сообщества

В современном мире разработки программного обеспечения сложно обойтись без работы с массивами данных. Как следствие актуальным является вопрос использования различных хранилищ данных. Исторически реляционные базы данных имели широкое применение, и, конечно, платформа .NET не могла не предоставлять свои инструменты для работы с ними.

Перед вами подробное руководство по одному из таких инструментов – Entity Framework Core. EF Core стал почти стандартом при разработке .NET-приложений, использующих реляционные базы данных, и большое число разработчиков успешно применяют его в своих проектах. EF Core позволяет легко начать работу и быстро реализовать простые сценарии по взаимодействию с базами данных. В первой части книги описываются основы фреймворка и вся необходимая информация, чтобы начать работать. Вместе с тем EF Core обеспечивает поддержку и более сложных сценариев. Во второй части более подробно раскрываются внутренние механизмы фреймворка и приводятся сведения о тонкой настройке EF Core, а в третьей части рассматриваются задачи, возникающие при использовании EF Core в реальных приложениях.

Книга будет интересна как новичкам, так и более опытным разработчикам, уже знакомым с основами EF Core. Автор подробно описал, как использовать EF Core, а также затронул большое количество смежных тем, которые помогут понять место фреймворка в экосистеме разработки на платформе .NET. Если у вас уже есть опыт работы с предыдущей версией Entity Framework 6, то вы найдете большое количество сравнений и описание отличий в поведении фреймворков. А если вам интересны нереляционные базы данных, то на примере Cosmos DB вы сможете узнать, как EF Core позволяет работать с такими хранилищами (включая разработку приложения с использованием CQRS-подхода).

Отдельная благодарность автору за «прикладные» главы книги. EF Core (как и многие другие ORM) прост в использовании, но применение его в сложных сценариях может повлечь за собой проблемы производительности. Автор посвятил этой проблеме отдельную главу, подробно разобрал основные проблемы производительности, методы их диагностики и решения. Также в книге затронуты вопросы проектирования (с использованием популярного подхода предметно-ориентированного проектирования, DDD) и тестирования приложений.

В итоге получилась всесторонняя книга о EF Core (и не только), которую можно смело рекомендовать всем, кто работает с базами данных на платформе .NET. Команда DotNet.Ru с удовольствием работала над переводом книги и благодарит автора за отличный материал. Приятного чтения!

Над переводом работали представители сообщества DotNet.Ru:

Игорь Лабутин;	Сергей Бензенко;	Дмитрий Жабин;	Радмир Тагиров;
Рустам Сафин;	Илья Лазарев;	Вадим Мингажев;	Алексей Ростов;
Евгений Буторин;	Андрей Беленцов;	Виталий Илюхин;	Анатолий Кулаков.

Отзывы на книгу «Entity Framework Core в действии»

Наиболее полный справочник по EF Core, который есть или когда-либо будет.

– *Стивен Бирн*, Intel Corporation

Полное руководство по EF Core. Это самый практичный способ улучшить свои навыки по работе с EF Core с примерами из реальной жизни.

– *Пол Браун*, Diversified Services Network

Я твердо верю, что любой, кто использует EF Core, найдет в этой книге что-то полезное для себя.

– *Энн Эпштейн*, Headspring

Остается для меня полезным ресурсом при работе с Entity Framework.

– *Фостер Хейнс*, J2 Interactive

Предисловие

Приходилось ли вам когда-нибудь работать над приложением, которое не использует данные и требует средств взаимодействия с хранилищем данных? За несколько десятилетий работы в качестве разработчика программного обеспечения каждое приложение, над которым я работала или помогала в работе другим, зависело от чтения и записи данных в хранилище определенного типа. Когда я стала индивидуальным предпринимателем в 1990-х г., то придумала для своей компании название Data Farm. Я определенно фанат данных.

За последние несколько десятилетий корпорация Microsoft прошла множество итераций фреймворков для доступа к хранящимся в базе данным. Если вы какое-то время работали в этой сфере, то, возможно, помните DAO и RDO, ADO и ADO.NET. В 2006 году Microsoft поделилась первыми версиями тогда еще не названного Entity Framework (EF) на основе работы, проделанной в Microsoft Research на закрытой встрече в TechEd. Я была одной из немногих, кого пригласили на эту встречу. Я впервые увидела инструмент объектно-реляционного отображения (Object Relational Mapper – ORM), библиотеку, цель которой – освободить разработчиков от излишней рутинной работы по созданию подключений и команд путем написания SQL-запросов, преобразования результатов запроса в объекты и преобразования изменений объекта в SQL, чтобы сохранить их в базе данных.

Многие из нас беспокоились, что это очередной фреймворк для доступа к хранящимся в базе данным, от которого Microsoft откажется в ближайшее время, заставив нас изучать еще один в будущем. Но история доказала, что мы ошибались. Пятнадцать лет спустя Microsoft по-прежнему инвестирует в Entity Framework, который превратился в кросс-платформенный Entity Framework Core с открытым исходным кодом и продолжает оставаться основной библиотекой Microsoft для доступа к данным для разработчиков .NET.

За 15 лет существования и развития EF эволюционировал и .NET. Возможности EF и EF Core стали более обширными, но в то же время, когда дело доходит до создания современных программных систем, эта библиотека стала умнее и понимает, когда нужно просто не мешать разра-

ботчику. Мы можем настраивать отображения для поддержки хранения со сложной схемой базы данных. Как специалист по предметно-ориентированному проектированию, я была очень довольна тем вниманием, которое команда уделила тому, чтобы позволить EF Core сохранять тщательно спроектированные сущности, объекты значений и агрегаты, которые от природы не наделены знанием о схеме базы данных.

Будучи одним из первых пользователей, в тесном сотрудничестве с командой EF еще до первого выпуска этой библиотеки я написала четыре книги по Entity Framework в период с 2008 по 2011 г. Хотя мне и в самом деле нравится писать, в конце концов я обнаружила, что мне также нравится создавать видео, поэтому я сосредоточила свои усилия на создании и публикации курсов по EF Core и другим темам в качестве автора на сайте Pluralsight. Я по-прежнему пишу статьи, но больше не пишу книг, поэтому очень счастлива, что Джон П. Смит нашел способ сотрудничества с издательством Manning и написал эту книгу.

Когда Джон опубликовал первое издание «*Entity Framework Core в действии*», я узнала в нем родственную душу, «любопытного кота», который приложил все возможные усилия в своем стремлении понять, как работает EF Core. Точно так же серьезно он относится к изложению этой информации, гарантируя, что его читатели не потеряют нить повествования и получат реальные знания. Поскольку я продолжала создавать учебные ресурсы для тех, кто предпочитает учиться по видео, мне было приятно порекомендовать работу Джона тем, кто ищет заслуживающее доверия издание по EF Core. Обновить содержимое книги, чтобы привести ее в соответствие с новейшей версией EF Core 5, – нелегкая задача. Джон снова заработал мое уважение (и уважение многих других людей), когда на свет появилось издание, которое вы сейчас держите в руках.

Благодаря этой книге вы получаете три книги в одной. Во-первых, Джон подскажет вам основы и даже создаст несколько простых приложений, использующих EF Core. Когда вы освоитесь, можно будет подробнее изучить использование EF Core на среднем уровне, применяя связи, миграции и управление, выходящее за рамки стандартного поведения EF Core. Наконец, придет время использовать EF Core в реальных приложениях, решая такие важные задачи, как производительность и архитектура. Тщательные исследования Джона и его собственный опыт работы с крупными программными приложениями делают его квалифицированным и заслуживающим доверия гидом.

— ДЖУЛИ ЛЕРМАН

Джули Лерман известна как ведущий эксперт по Entity Framework и EF Core за пределами Microsoft. Она является автором серии книг Programming Entity Framework и десятков курсов на сайте Pluralsight.com. Джули обучает компании, как проводить модернизацию программного обеспечения. Ее можно встретить на конференциях, посвященных программному обеспечению в разных уголках света, где она выступает с докладами по EF, предметно-ориентированному программированию и другим темам.

Введение

Любой разработчик программного обеспечения должен привыкать к необходимости изучения новых библиотек или языков, но для меня это обучение было немного экстремальным. Я перестал заниматься программированием в 1988 г., когда перешел в технический менеджмент, и не возвращался к нему до 2009 г. – перерыв в 21 год. Сказать, что ландшафт изменился, – не сказать ничего; я чувствовал себя ребенком в рождественское утро с таким количеством прекрасных подарков, что не мог взять их все.

Поначалу я совершал все ошибки, присущие новичку, например я думал, что объектно-ориентированное программирование – это использование наследования, однако это не так. Но я изучил новый синтаксис и новые инструменты (вау!) и наслаждался объемом информации, который мог получить в интернете. Я решил сосредоточиться на стеке Microsoft, в основном по причине того, что по нему было доступно большое количество документации. В то время это был хороший выбор, но с выходом .NET Core с открытым исходным кодом и многоплатформенным подходом я понял, что это был отличный выбор.

Первые приложения, над которыми я работал в 2009 г., оптимизировали и отображали потребности здравоохранения с географической точки зрения, особенно с точки зрения расположения лечебных центров. Эта задача требовала сложной математики (этим занималась моя жена) и серьезной работы с базами данных. Я прошел через ADO.NET и LINQ to SQL. В 2013 г. я переключился на Entity Framework (EF), когда EF 5 поддержал пространственные (географические) типы SQL, а затем перешел на EF Core сразу после его выпуска.

За прошедшие годы я часто использовал EF Core и в клиентских проектах, и для создания библиотек с открытым исходным кодом. Помимо этой книги, я много писал о EF Core в собственном блоге (www.thereformedprogrammer.net). Оказывается, мне нравится брать сложные идеи и пытаться сделать так, чтобы их легче было понять другим. Надеюсь, мне удастся сделать это и в данной книге.

«*Entity Framework Core в действии*» охватывает все функции EF Core 5.0 со множеством примеров и кодом, который вы можете за-

пустить. Кроме того, я включил сюда много паттернов и практик, которые помогут вам создать надежный и поддающийся рефакторингу код. В третьей части книги, которая называется «Использование Entity Framework Core в реальных приложениях», показаны создание и доставка реальных приложений. И у меня есть не одна, а три главы о настройке производительности EF Core, поэтому у вас под рукой множество методов повышения производительности, когда ваше приложение работает не так хорошо, как вам нужно.

Одними из самых приятных для написания глав были главы, посвященные тому, как EF Core работает внутри (главы 1, 6 и 11), и настройке производительности приложения (главы 14, 15 и 16). Лично я многому научился, используя модульную монолитную архитектуру (глава 13) и создавая полноценное приложение с помощью Cosmos DB (глава 16). Попутно я стараюсь представить плюсы и минусы каждого используемого мной подхода, поскольку не верю, что в программном обеспечении есть такое понятие, как «серебряная пуля». Есть лишь ряд компромиссов, которые мы, будучи разработчиками, должны учитывать при выборе того, как реализовать что-либо.

Благодарности

Хотя бóльшую часть работы над книгой проделал я, мне очень помогли и другие люди, и я хочу поблагодарить их всех.

Спасибо моей жене, доктору Хоноре Смит, за то, что она терпела, как я три четверти года сидел перед компьютером, и за то, что вернула меня к программированию. Я люблю ее до потери сознания. Еще одна особая благодарность моему большому другу JC за помощь и поддержку.

Работать с Manning Publications было восхитительно. Это был надежный и всеобъемлющий процесс, трудоемкий, но продуманный, обеспечивающий в итоге отличный продукт. Команда была просто замечательная, и я хочу перечислить значимых лиц в хронологическом порядке, начиная с Брайана Сойера, Брекина Эли, Марины Майклс, Джоэля Котарски, Рейханы Марканович, Йосипа Мараса, Хизер Такер, Александара Драгосавлевича и многих других, кто помогал в выпуске книги. Марина Майклс была моим основным контактным лицом во время первого издания, и, очевидно, я не доставил ей слишком много проблем, поскольку она любезно согласилась мне помочь со вторым изданием.

Кроме того, мне очень помогла загруженная команда EF Core. Помимо ответов на многочисленные вопросы, которые были подняты в репозитории EF Core на GitHub, они проверили несколько глав. Особого упоминания заслуживают Артур Викерс и Шай Роянски за рецензирование некоторых глав. Остальные члены команды перечислены в алфавитном порядке: Андрей Свирид, Брайс Лэмбсон, Джереми Ликнесс, Маурициий Марковски и Смит Пател.

Я также хотел бы поблагодарить Жюльена Похи, технического корректора, и рецензентов: Эла Пезевски, Анну Эпштейн, Фостера Хейнса, Хари Хальса, Янека Лопеса, Джеффа Ноймана, Джоэля Клермона, Джона Роудса, Мауро Кверчиоли, Пола Г. Брауна, Раушана Джа, Рикардо Переса, Шона Лэма, Стивена Бирна, Сумита К Сингха, Томаса Гета, Томаса Оверби Хансена и Уэйна Мэзер. Ваши предложения помогли сделать эту книгу лучше.

Об этой книге

Книга «*Entity Framework Core в действии*» посвящена быстрому и правильному написанию кода работы с базой данных с помощью EF Core, чтобы в конечном итоге обеспечить высокую производительность. Чтобы помочь с такими аспектами, как «быстро и правильно», я включил сюда большое количество примеров со множеством советов и приемов. Попутно я немного расскажу, как EF Core работает изнутри, потому что эта информация поможет вам, когда что-то работает не так, как, по вашему мнению, должно работать.

У Microsoft неплохая документация, но в ней нет места для подробных примеров. В этой книге я постараюсь дать вам хотя бы один пример по каждой функции, о которой я рассказываю, а в репозитории на GitHub часто можно будет найти модульные тесты (см. раздел «О коде», где есть ссылки), которые тестируют функцию несколькими способами. Иногда чтение модульного теста может показать, что происходит, гораздо быстрее, нежели чтение текста в книге, поэтому считайте модульные тесты полезным ресурсом.

Кому адресована эта книга?

Эта книга предназначена как для разработчиков программного обеспечения, которые никогда раньше не использовали EF, так и для опытных разработчиков EF Core, а также для всех, кто хочет знать, на что способен EF Core. Я предполагаю, что вы знакомы с разработкой в .NET на C# и имеете хоть какое-то представление о том, что такое реляционная база данных. Не нужно быть экспертом в C#, но если вы новичок, возможно, вам будет трудно читать некоторые части кода, поскольку я не объясняю C#. Книга начинается с основных команд EF Core, которые должны быть доступны большинству программистов на C#, но начиная со второй части темы становятся более сложными по мере углубления в функции EF Core.

Как устроена эта книга

Я попытался построить маршрут, который начинается с основ (часть I), после чего мы переходим к деталям (часть II), а заканчивается он полезными инструментами и методами (часть III). Я не предполагаю, что вы прочитаете эту книгу от корки до корки, особенно справочный раздел из второй части, но хотя бы беглое чтение первых шести глав поможет вам понять основы, которые я использую позже.

Часть I «Начало»:

- глава 1 знакомит вас с суперпростым консольным приложением, использующим EF Core, чтобы вы могли увидеть все части EF Core в действии. Кроме того, я привожу обзор того, как работает EF Core и для чего можно его использовать;
- в главе 2 рассматривается запрос (чтение данных) к базе данных. Я расскажу о связях между данными, хранящимися в базе, и о том, как загрузить эти связанные данные с помощью EF Core;
- в главе 3 мы переходим к изменению данных в базе: добавлению новых данных, обновлению существующих данных и их удалению;
- в главе 4 рассматриваются различные способы построения надежной бизнес-логики, использующей EF Core для доступа к базе данных. *Бизнес-логикой* называется код, реализующий бизнес-правила или рабочий процесс для конкретной бизнес-задачи, которую решает ваше приложение;
- глава 5 посвящена созданию приложения ASP.NET Core, использующего EF Core. Она объединяет код, разработанный в главах 2, 3 и 4, для создания веб-приложения. Помимо этого, я рассказываю о развертывании веб-приложения и доступе к размещенной базе данных;
- глава 6 охватывает широкий круг тем. Большинство из них содержит описание одного из аспектов EF Core в сочетании со способами использования этой функции в коде.

Часть II «Об Entity Framework Core в деталях»:

- в главе 7 рассматривается настройка нереляционных свойств – свойств, содержащих значение, например `int`, `string`, `DateTime` и т. д.;
- в главе 8 рассматривается настройка связей между классами, например классом `Book`, связанным с одним или несколькими классами `Author`. Кроме того, она включает в себя специальные методы отображения, например отображение нескольких классов в одну таблицу;
- в главе 9 описаны все способы изменения структуры базы данных при использовании EF Core, а также рассматриваются проблемы, возникающие, когда вам нужно изменить структуру базы данных, используемой работающим приложением;
- в главе 10 рассматриваются расширенные функции сопоставления и вся область обнаружения и обработки конфликтов параллельного доступа;

- в главе 11 подробно рассмотрено, как работает DbContext EF Core, с подробным описанием того, что различные методы и свойства делают внутри DbContext-приложения.

Часть III «Использование Entity Framework Core в реальных приложениях»:

- в главе 12 представлены два подхода к отправке сообщений расширенным методам `SaveChanges` и `SaveChangesAsync`. Эти подходы предоставляют еще один способ объединения нескольких обновлений в одно транзакционное обновление базы данных;
- в главе 13 рассматривается применение предметно-ориентированного проектирования (DDD) к классам, отображаемым в базу данных с помощью EF Core, а также описывается еще один архитектурный подход, используемый в версии приложения Book App из части III;
- в главе 14 перечислены все проблемы, которые могут повлиять на производительность доступа к базе данных, и обсуждается, что с ними делать;
- глава 15 представляет собой рабочий пример настройки производительности приложения EF Core. Я беру исходный запрос на отображение приложения Book App, разработанного в части I, и применяю три уровня настройки производительности;
- в главе 16 для дальнейшей настройки приложения Book App используется Cosmos DB, что раскрывает сильные и слабые стороны этой базы данных и провайдера EF Core для нее. В конце главы рассказывается, что нужно делать при переходе с одного типа базы данных на другой;
- глава 17 посвящена модульному тестированию приложений, использующих EF Core. Кроме того, я создал пакет NuGet, который вы можете использовать для упрощения своих модульных тестов.

Приложение:

- приложение A знакомит с языком LINQ, используемым в EF Core. Это приложение полезно для тех, кто незнаком с LINQ или хочет быстро вспомнить его.

О коде

Мне кажется, что я действительно что-то знаю только в том случае, если я написал код для использования этой функции или возможности, поэтому вам доступен репозиторий GitHub на странице http://mng.bz/XdlG.

ПРИМЕЧАНИЕ Я настоятельно рекомендую клонировать код с вышеуказанной страницы. У копии репозитория, указанной на странице книг Manning, имеется проблема с веткой `Part3` из-за длинных имен каталогов.

Этот репозиторий содержит код приложений, которые я показываю в книге, и модульные тесты, которые я запускал, чтобы убедиться, что все, о чем я говорю в книге, правильно. У этого репозитория три ветки:

- `master`, охватывающая первую часть книги (главы 1–6);
- `Part2`, охватывающая вторую часть книги (главы 7–11);
- `Part3`, охватывающая третью часть книги (главы 12–17).

Чтобы запустить любое из приложений, сначала необходимо прочитать файл Readme на странице http://mng.bz/yYjG в репозитории GitHub. Файл Readme каждой ветки состоит из трех основных разделов:

- *что нужно установить для запуска примеров приложений*, где указаны приложения для разработки, версия .NET и требования к базе данных для запуска приложений из репозитория GitHub (эта информация одинакова для всех веток);
- *что можно запустить в этой ветке*, где указано, какое приложение (приложения) можно запустить в выбранной вами ветке репозитория GitHub;
- *как найти и запустить модульные тесты*, где говорится, где находятся модульные тесты и как их запустить.

По мере прохождения трех частей книги вы можете выбирать каждую ветку, чтобы получить доступ к коду, предназначенному именно для этой части. Также обратите внимание на связанные модульные тесты, сгруппированные по главам и функциям.

ПРИМЕЧАНИЕ В главе 17, посвященной модульному тестированию, я использовал созданную мной библиотеку. Эта библиотека, которую можно найти на странице https://github.com/JonPSmith/EfCore.TestSupport, – обновленная версия созданной мной библиотеки EfCore.TestSupport для первого издания данной книги. Теперь здесь используются новые функции, доступные в EF Core 5. Это библиотека с открытым исходным кодом (лицензия MIT), поэтому вы можете использовать пакет NuGet под названием EfCore.TestSupport (версия 5 и новее) в собственных модульных тестах.

Соглашения об оформлении программного кода

Примеры кода в этой книге и их вывод написаны моноширинным шрифтом и часто сопровождаются аннотациями. Примеры намеренно делаются максимально простыми, потому что они не представляют собой части для повторного использования, которые можно вставить в ваш код. Образцы кода урезаны, чтобы вы могли сосредоточиться на проиллюстрированном принципе.

Эта книга содержит множество примеров исходного кода, как в пронумерованных листингах, так и в самом тексте. В обоих случаях исходный код отформатирован с использованием моноширинного шрифта,

`как этот`, чтобы отделить его от остального текста. Кроме того, иногда используется **`жирный шрифт`**, чтобы выделить код, который изменился по сравнению с предыдущими шагами в главе, например когда в существующую строку кода добавляется новая функция.

Во многих случаях исходный код был переформатирован; мы добавили разделители строк и переделали отступы, чтобы уместить их по ширине книжных страниц. Также из многих листингов, описываемых в тексте, мы убрали комментарии. Некоторые листинги сопровождают аннотации, выделяющие важные понятия.

Исходный код примеров из этой книги доступен для скачивания из репозитория GitHub (http://mng.bz/XdlG).

Автор онлайн

Приобретая книгу «*EF Core в действии*», вы получаете бесплатный доступ на приватный веб-форум издательства Manning Publications, где можно оставить отзывы о книге, задать технические вопросы и получить помощь от авторов и других пользователей. Чтобы получить доступ к форуму, откройте в браузере страницу https://livebook.manning.com/book/entity-framework-core-in-action-second-edition. На странице https://livebook.manning.com/#!/discussion можно получить дополнительную информацию о форумах Manning и правилах поведения на них.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угас!

Форум и архивы предыдущих дискуссий будут оставаться доступными, пока книга продолжает издаваться.

Онлайн-ресурсы

Полезные ссылки на документацию Microsoft и код:

- *документация Microsoft по EF Core* – https://docs.microsoft.com/en-us/ef/core/;
- *код EF Core* – https://github.com/dotnet/efcore;
- *ASP.NET Core, работа с EF Core* – https://docs.microsoft.com/en-us/aspnet/core/data/;
- *тег EF Core на Stack Overflow* `[entity-framework-core]` – https://stackoverflow.com.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понра-

вилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Джон П. Смит – внештатный разработчик программного обеспечения и программный архитектор, специализирующийся на .NET Core и Azure. Он работает в основном над серверной частью клиентских приложений, обычно используя Entity Framework Core (EF Core) и веб-приложения ASP.NET Core. Джон работает удаленно с клиентами по всему миру, многие из его проектов из Соединенных Штатов. Обычно он помогает с проектированием, настройкой производительности и написанием разделов клиентского приложения.

Джон интересуется определением паттернов и созданием библиотек, которые повышают скорость разработки приложений при использовании EF Core и ASP.NET Core. Его библиотеки были написаны, потому что он нашел некую повторяющуюся часть проекта, над которым работал, которую можно было превратить в полезную библиотеку. Сводку его основных библиотек можно увидеть на странице в GitHub (https://github.com/JonPSmith).

Кроме того, Джон ведет собственный технический блог на странице http://www.thereformedprogrammer.net, где рассматривает темы, связанные с EF Core, ASP.NET Core и различными архитектурными подходами. Самая популярная статья в его блоге посвящена улучшенной системе авторизации ASP.NET Core; см. http://mng.bz/ao2z. Он выступал на нескольких конференциях и митапах в Великобритании.

Об изображении на обложке

Иллюстрация на обложке книги называется «Жена франкского торговца». Она взята из книги Томаса Джефферисса «Коллекция платьев разных народов, древних и современных» (четыре тома), Лондон, изданной между 1757 и 1772 г. На титульном листе указано, что эти иллюстрации представляют собой гравюры на медной пластине ручной раскраски, усиленные гуммиарабиком.

Томаса Джефферисса (1719–1771) называли «географом короля Георга III». Он был английским картографом и ведущим поставщиком карт своего времени. Гравировал и печатал карты для правительства и других официальных организаций, а также выпустил широкий спектр коммерческих карт и атласов, особенно Северной Америки. Его работа в качестве картографа вызвала у него интерес к местным обычаям одежды на землях, которые он исследовал и наносил на карту и которые с блеском представлены в этой коллекции. Очарование далеких стран и путешествия ради удовольствия были относительно новым явлением в конце XVIII века, и коллекции, подобные этой, были популярны, знакомя и туриста, и путешественника, сидящего в кресле, с жителями других стран.

Разнообразие рисунков в томах Джефферисса наглядно свидетельствует об уникальности и индивидуальности народов мира около 200 лет назад. Манеры одеваться сильно изменились с тех пор, а своеобразие каждого региона, такое яркое в то время, давно поблекло. Сейчас бывает трудно различить обитателей разных континентов. Возможно, если смотреть на это оптимистично, мы обменяли культурное и визуальное разнообразие на более разнообразную частную жизнь или более разнообразную интеллектуальную и техническую жизнь.

В то время когда сложно отличить одну компьютерную книгу от другой, мы в Manning высоко ценим изобретательность, инициативу и, конечно, радость от компьютерного бизнеса, используя книжные обложки, основанные на разнообразии жизни в разных регионах два века назад, которое оживает благодаря картинкам из этой коллекции.

Часть I

Начало

Данные используются повсеместно, и каждый год их объем растет петабайтами. Значительная их часть хранится в базах данных. Кроме того, существуют миллионы приложений – в начале 2021 г. насчитывалось 1,2 млрд сайтов, – и большинству из них требуется доступ к данным. И это не говоря об интернете вещей. Поэтому неудивительно, что согласно прогнозу ведущей исследовательской и консалтинговой компании Gartner в 2021 г. глобальные расходы на IT достигнут 3,7 трлн долларов (http://mng.bz/gonl).

Хорошая новость для вас заключается в том, что ваши навыки будут востребованы. Однако плохая состоит в том, что потребность в быстрой разработке приложений неумолимо растет. Эта книга посвящена инструменту, который можно использовать для быстрого написания кода доступа к базам данных: Microsoft Entity Framework Core (EF Core). EF Core предоставляет объектно-ориентированный способ доступа к реляционным и нереляционным (NoSQL) базам данных в среде .NET. Самое замечательное в EF Core и других библиотеках .NET Core заключается в их быстродействии, а также в том, что они могут работать на платформах Windows, Linux и Apple.

В первой части книги мы с вами сразу же погрузимся в код. В главе 1 вы создадите очень простое консольное приложение, а к концу главы 5 – достаточно сложное веб-приложение для продажи книг. В главах 2 и 3 объясняется чтение и запись данных в реляционную базу данных, а в главе 4 рассказывается о написании бизнес-логики. В главе 5 вы будете использовать веб-фреймворк Microsoft ASP.NET Core для создания сайта по продаже книг. Глава 6 расширит ваши познания о внутренней работе EF Core с помощью ряда полезных ме-

тодов решения проблем с базами данных, таких как быстрый способ копирования данных в базу.

В первой части вам предстоит многое изучить, несмотря на то что я опущу здесь несколько тем, полагаясь на используемые по умолчанию настройки EF Core. Тем не менее эта часть должна дать вам четкое представление о том, что может делать этот фреймворк. Последующие части дополнят ваши познания, рассказывая о дополнительных функциях EF Core, с более подробной информацией о его настройках, а некоторые главы посвящены специфическим областям, таким как настройка производительности.

1 Введение в Entity Framework Core

В этой главе рассматриваются следующие темы:

- анатомия приложения EF Core;
- доступ к базе данных и ее обновление с помощью EF Core;
- изучение реального приложения EF Core;
- принятие решения об использовании EF Core в своем приложении.

Entity Framework Core, или *EF Core* – это библиотека, которую разработчики программного обеспечения могут использовать для доступа к базам данных. Есть много способов создать такую библиотеку, но EF Core разработана как инструмент объектно-реляционного отображения (ORM). Эти инструменты занимаются отображением между двумя мирами: реляционной базой данных (с собственным API) и объектно-ориентированным миром классов кода программного обеспечения. Основное преимущество EF Core – предоставить разработчикам программного обеспечения возможность быстрого написания кода доступа к базе данных на языке, который вы, возможно, знаете лучше, чем SQL.

EF Core поддерживает несколько платформ: он может работать в Windows, Linux и Apple. Это делается в рамках платформы .NET Core – отсюда и слово *Core* в названии. .NET 5 охватывает весь спектр настольных компьютеров, сеть, облачные и мобильные приложения, игры, интернет вещей (IoT) и искусственный интеллект (AI), однако эта книга посвящена EF Core.

EF Core – это логическое продолжение более ранней версии Entity Framework, известной как *EF6.x*. EF Core берет свое начало из многолетнего опыта, накопленного в предыдущих версиях, с 4 по 6.x. Он сохранил тот же тип интерфейса, что и EF6.x, но в нем есть существенные изменения, например возможность работы с нереляционными базами данных, чего не было предусмотрено в EF6.x. Я использовал EF5 и EF6 во многих приложениях до появления EF Core, что позволило мне увидеть значительные улучшения EF Core по сравнению с EF6.x как в отношении функциональных возможностей, так и в отношении производительности.

Эта книга предназначена как для разработчиков программного обеспечения, которые уже используют EF Core, так и для тех, кто никогда ранее не работал с Entity Framework, а также опытных разработчиков на EF6.x, которые хотят перейти на EF Core. Я предполагаю, что вы знакомы с разработкой на .NET с использованием C# и имеете представление о том, что такое реляционные базы данных. Возможно, вы не знаете, как писать запросы с помощью языка структурированных запросов (SQL), который используется большинством реляционных баз данных, потому что EF Core может проделать бóльшую часть этой работы за вас. Тем не менее я буду демонстрировать SQL запросы, которые создает EF Core, потому что это помогает понять, что происходит и как это работает. Для использования некоторых расширенных возможностей EF Core требуются знания SQL, но в книге есть большое количество диаграмм, которые помогут вам разобраться.

> **СОВЕТ** Если вы плохо разбираетесь в SQL и хотите узнать больше, предлагаю посетить онлайн-ресурс W3Schools: https://www.w3schools.com/sql/sql_intro.asp. Набор SQL-команд довольно обширен, но запросы, формируемые EF Core, используют лишь небольшое их подмножество (например, SELECT, WHERE и INNER JOIN), поэтому данный ресурс – хорошее место, чтобы получить базовые знания о SQL.

В этой главе вы познакомитесь с EF Core на примере небольшого приложения, которое работает с данной библиотекой. Заглянете под ее капот, чтобы увидеть, как команды на языке C# преобразуются в SQL запросы к базе данных. Обзорное представление того, что происходит внутри EF Core, поможет вам при чтении остальной части книги.

1.1 Что вы узнаете из этой книги

Книга содержит введение в библиотеку EF Core, начиная с основ и заканчивая более сложными ее аспектами. Чтобы извлечь максимальную пользу из книги, вы должны уметь разрабатывать приложения на C#, в том числе создавать проекты и загружать NuGet пакеты. Вы узнаете:

- основные принципы использования EF Core для работы с базами данных;
- как использовать EF Core в веб-приложении ASP.NET Core;
- многочисленные способы настройки EF Core, чтобы все работало именно так, как вам нужно;
- как использовать некоторые расширенные возможности баз данных;
- как управлять изменениями в схеме базы данных по мере роста приложения;
- как повысить производительность взаимодействия вашего кода с базой данных;
- и самое главное, как убедиться, что ваш код работает правильно.

На протяжении всей книги я буду создавать простые, но полнофункциональные приложения, чтобы вы могли увидеть, как EF Core работает в реальных ситуациях. Все эти приложения, а также множество советов и приемов, которые я собрал, занимаясь как рабочими, так и своими собственными проектами, доступны в репозитории с примерами.

1.2 Мой «момент озарения»

Прежде чем перейти к сути, позвольте рассказать вам об одном решающем моменте, который возник, когда я работал с Entity Framework, и который наставил меня на путь «поклонника» EF. Именно моя жена вернула меня в программирование после 21-летнего перерыва (это уже отдельная история!).

Моя жена, доктор Хонора Смит, преподает математику в Университете Саутгемптона и специализируется на моделировании систем здравоохранения, уделяя особое внимание расположению медицинских учреждений. Я работал с ней над созданием нескольких приложений для географического моделирования и визуализации для Национальной службы здравоохранения Великобритании, а также в Южной Африке над оптимизацией тестирования на ВИЧ/СПИД.

В начале 2013 г. я решил создать веб-приложение специально для моделирования здравоохранения. Я использовал только что появившиеся ASP.NET MVC4 и EF5, которые поддерживали пространственные типы SQL, обрабатывающие географические данные. Проект был успешным, но это была тяжелая работа. Я знал, что интерфейс будет сложным; это было одностраничное приложение, где использовалась библиотека Backbone.js, но я был удивлен тем, сколько времени мне потребовалось, чтобы выполнить работу на стороне сервера.

Я применил передовые методы работы с программным обеспечением и убедился, что база данных и бизнес-логика соответствуют задаче – моделированию и оптимизации расположения медицинских учреждений. Все было нормально, но я потратил слишком много времени на написание кода для преобразования записей базы данных и бизнес-

логики в форму, подходящую для отображения пользователю. Кроме того, я использовал паттерны «Репозиторий» (Repository) и «Единица работы» (Unit of Work), чтобы скрыть код EF5, и мне постоянно приходилось настраивать области, чтобы репозиторий работал правильно.

В конце проекта я всегда оглядываюсь назад и спрашиваю: «Мог бы я сделать это лучше?» Будучи разработчиком программного обеспечения, я всегда ищу части, которые (а) хорошо работали, (б) повторялись и должны быть автоматизированы или (с) имели постоянные проблемы. На этот раз список выглядел так:

- *работает хорошо* – ServiceLayer, слой в моем приложении, который изолировал/адаптировал нижние уровни приложения от внешнего интерфейса ASP.NET MVC4, работал хорошо (я представлю эту многоуровневую архитектуру в главе 2);
- *повторяемость* – я использовал классы ViewModel, также известные как *объекты передачи данных* (DTO), чтобы представить данные, которые мне нужно было показать пользователю. Все прошло хорошо, но написание кода для копирования таблиц базы данных во ViewModel и DTO было однообразным и скучным делом (о ViewModels и DTO рассказывается в главе 2);
- *наличие постоянных проблем* – паттерны «Репозиторий» и «Единица работы» мне не подошли. На протяжении всего проекта возникали постоянные проблемы (я расскажу о паттерне «Репозиторий» и его альтернативах в главе 13).

В результате своего обзора я создал библиотеку GenericServices (https://github.com/JonPSmith/GenericServices) для использования с EF6.x. Она автоматизировала копирование данных между классами базы данных и классами ViewModel и DTO, устраняя необходимость в использовании вышеуказанных паттернов. Библиотека вроде бы работала хорошо, но, чтобы провести для нее стресс-тест, я решил создать интерфейс на основе одного из примеров баз данных Microsoft: AdventureWorks 2012 Lite. Я создал все приложение с помощью библиотеки для разработки пользовательских интерфейсов за 10 дней!

Entity Framework + правильные библиотеки + правильный подход = быстрая разработка кода доступа к базе данных

Сайт был не очень красивым, но дело не во внешнем виде. Анализируя то, как я использовал паттерны «Репозиторий» и «Единица работы» с EF6.x, я нашел лучший подход. Затем, инкапсулируя этот подход в библиотеку GenericServices, я автоматизировал процесс создания команд Create, Read, Update и Delete (CRUD). Результат позволил мне очень быстро создавать приложения – определенно это был «момент озарения», и я «подсел» на EF.

С тех пор я создал новые библиотеки, работающие с EF Core, которые, как я обнаружил, значительно ускоряют разработку 90 % обращений к базе данных. Я работаю разработчиком по контракту, и эти библиотеки с открытым исходным кодом, доступные и вам, автома-

тизируют некоторые стандартные требования, позволяя сосредоточиться на более сложных темах, таких как понимание потребностей клиента, написание пользовательской бизнес-логики и настройка производительности там, где это необходимо. Я расскажу об этих библиотеках в последующих главах.

1.3 Несколько слов для разработчиков EF6.x

> **ЧТОБЫ СЭКОНОМИТЬ ВРЕМЯ** Если вы не работали с Entity Framework 6.x, то можете пропустить этот раздел.

Если вы работали с EF6.x, то многое из EF Core будет вам знакомо. Чтобы помочь вам сориентироваться в этой книге, я добавил примечания по EF6.

> **EF6** Обращайте внимание на эти примечания в ходе чтения. Они указывают на те моменты, где EF Core отличается от EF6.x. Также обязательно просмотрите аннотации в конце каждой главы, которые указывают на наиболее серьезные отличия между EF6 и EF Core.

Кроме того, я дам вам один совет из своего путешествия по EF Core. Я хорошо знаю EF6.x, но эти знания стали проблемой, когда я начал использовать EF Core. Я применял к проблемам подход на базе EF6.x и не заметил, что у EF Core появились новые способы их решения. В большинстве случаев подходы схожи, а где-то нет.

Мой совет вам как существующему разработчику EF6.x – подходите к EF Core как к новой библиотеке, которая была написана для имитации EF6.x, но учтите, что она работает иначе. Таким образом, вы будете готовы к новым способам работы в EF Core.

1.4 Обзор EF Core

Можно использовать EF Core как инструмент объектно-реляционного отображения (далее: ORM), который работает с реляционной базой данных и миром классов и программного кода .NET. Смотрите табл. 1.1.

Таблица 1.1. Как используется EF Core для работы с базой данных в .NET

Реляционная база данных	Программное обеспечение .NET
Таблица	Класс .NET
Столбцы таблицы	Свойства/поля класса
Записи/строки в таблице	Элементы в коллекциях .NET, например `List`
Первичные ключи: уникальная запись/строка	Уникальный экземпляр класса
Внешние ключи: определение связи	Ссылка на другой класс
SQL-оператор, например `WHERE`	Запросы на языке LINQ, например `Where(p => …`

1.4.1 Недостатки инструментов объектно-реляционного отображения

Создать хороший инструмент ORM сложно. Хотя EF6.x или EF Core могут показаться простыми в использовании, иногда «магия» EF Core может застать вас врасплох. Позвольте упомянуть две проблемы, о которых следует знать, прежде чем подробно рассматривать, как работает EF Core.

Первая проблема – это *объектно-реляционное несоответствие*. Серверы баз данных и объектно-ориентированное программное обеспечение используют разные принципы; базы данных используют первичные ключи для определения уникальности строки, тогда как экземпляры классов .NET по умолчанию считаются уникальными по их ссылке. EF Core обрабатывает бóльшую часть несоответствия за вас, но классы .NET получают первичные и внешние ключи, которые являются дополнительными данными, необходимыми только для базы данных. Программная версия классов не нуждается в этих дополнительных свойствах, а базе данных они нужны.

Вторая проблема заключается в том, что инструмент ORM – в особенности такой всеобъемлющий, как EF Core, – является противоположностью первой проблемы. EF Core настолько хорошо «скрывает» базу данных, что иногда можно забыть о том, что находится внутри нее. Эта проблема может привести к написанию кода, который будет хорошо работать в C#, но не подходит для базы данных. Один из примеров – вычисляемое свойство, возвращающее полное имя человека, путем объединения свойств `FirstName` и `LastName` в классе, например

```
public string FullName => $"{FirstName} {LastName}";
```

Это свойство является правильным в C#, но это же свойство вызовет исключение, если вы попытаетесь выполнить фильтрацию или сортировку по нему, потому что EF Core требуется столбец FullName в таблице, чтобы можно было применить SQL-команду `WHERE` или `ORDER` на уровне базы данных.

Вот почему в этой главе я показываю, как EF Core работает изнутри и какой SQL он создает. Чем больше вы понимаете, что делает EF Core, тем лучше вы будете подготовлены для написания правильного кода и, что более важно, будете знать, что делать, если ваш код не работает.

ПРИМЕЧАНИЕ На протяжении всей книги я использую подход «Сделайте так, чтобы это работало, но будьте готовы сделать это быстрее, если нужно». EF Core позволяет быстро осуществлять разработку, но я знаю, что из-за EF Core или из-за того, что я плохо его использовал, производительность моего кода доступа к базе данных может быть недостаточно хорошей для определенной бизнес-потребности. В главе 5 рассказывается, как изолировать EF Core, чтобы можно было выполнить настройку с минимальными побочными эффектами, а в главе 15

показано, как найти и улучшить код базы данных, который работает недостаточно быстро.

1.5 Что насчет нереляционных (NoSQL) баз данных?

Нельзя говорить о реляционных базах данных, не упомянув нереляционные базы данных, также известные в обиходе как NoSQL (http://mng.bz/DW63). И реляционные, и нереляционные базы данных играют важную роль в современных приложениях. Я использовал Microsoft SQL Server (реляционная база данных) и Azure Tables (нереляционная база данных) в одном приложении для удовлетворения двух бизнес-требований.

EF Core работает и с реляционными, и нереляционными базами данных – в отличие от EF6.x, которая была разработана только для реляционных баз данных. Большинство команд EF Core, описанных в этой книге, применимы к обоим типам баз данных, но есть некоторые различия на уровне базы данных, которые не учитывают некоторые более сложные команды для обеспечения масштабируемости и производительности.

В EF Core 3.0 добавлен провайдер базы данных Azure, Cosmos DB, о котором я расскажу в главе 16. В этой главе я указываю на различия между реляционной базой данных и Cosmos DB. Я был удивлен тем, что обнаружил. Теперь, когда в EF Core были внесены изменения, чтобы его можно было использовать для работы с нереляционными базами данных, я ожидаю, что для этого типа БД будут написаны и другие провайдеры.

> **ПРИМЕЧАНИЕ** У Cosmos DB и других нереляционных баз данных есть много сильных сторон по сравнению с реляционными базами данных. Например, гораздо проще иметь несколько копий нереляционных БД по всему миру, что позволяет пользователю быстрее получить доступ, а если центр обработки данных выйдет из строя, то другие копии могут принять на себя нагрузку. Но у нереляционных баз данных также есть некоторые ограничения по сравнению с базами данных SQL; прочтите главу 16, где приводится углубленный анализ преимуществ и ограничений Cosmos DB.

1.6 Ваше первое приложение, использующее EF Core

В этой главе мы начнем с простого примера, чтобы можно было сосредоточиться на том, что делает EF Core, а не код. Мы напишем не-

большое консольное приложение MyFirstEfCoreApp, которое обращается к простой базе данных. Задача приложения – вывести список книг и обновить его в прилагаемой базе данных. На рис. 1.1 показан вывод консоли.

Рис. 1.1 Консольное приложение предоставляет команду, которая использует запрос EF Core для чтения и отображения всех книг из вашей базы данных, а также команду для обновления. Эти две команды показывают, как EF Core работает изнутри

Это приложение не получит никаких призов за свой интерфейс или сложность, но это хороший способ начать работу, особенно потому, что я хочу показать вам, как EF Core работает изнутри, чтобы понять, о чем пойдет речь в последующих главах.

Пример приложения можно скачать из репозитория Git на странице http://mng.bz/XdlG. Вы можете посмотреть код и запустить приложение. Для этого вам понадобятся инструменты разработчика.

1.6.1 Что нужно установить

У Microsoft есть два редактора для работы с приложениями .NET Core: Visual Studio и Visual Studio Code (сокращенно VS Code). Visual Studio немного проще в использовании, и если вы новичок в .NET, то предлагаю использовать его. Его можно скачать на сайте www.visualstudio.com. Существует множество версий, в том числе и бесплатная Community версия, но необходимо прочитать лицензию, чтобы убедиться, что вы соответствуете требованиям: www.visualstudio.com/vs/community.

При установке Visual Studio в Windows обязательно включите функцию кросс-платформенной разработки (Cross-Platform Development) .NET Core и функцию хранения и обработки данных (Data storage and processing), которые находятся в разделе **Other Toolsets** (Другие наборы инструментов) на этапе **Install Workloads** (Установка рабочих нагрузок). Выбрав функцию кросс-платформенной разработки, вы

также установите в своей системе комплект для разработки программного обеспечения .NET Core, который необходим для создания приложений на .NET. Посетите страницу http://mng.bz/2x0T для получения дополнительной информации.

Если вы хотите использовать бесплатный редактор VS Code, то можете скачать его на странице https://code.visualstudio.com. Вам нужно будет выполнить дополнительные настройки в системе, например установить последнюю версию .NET Core SDK и SQL Server Express LocalDB. Как я уже сказал, если вы новичок в .NET, то предлагаю использовать Visual Studio для Windows, так как она может многое настроить за вас.

Одна из версий Visual Studio работает на компьютере с Apple Macintosh, а версии VS Code работают в Windows, на Mac и в Linux. Если вы хотите запустить какое-либо приложение или модульный тест, то на вашем компьютере должен быть установлен SQL Server. Возможно, вам потребуется изменить имя сервера в строках подключения.

Можно запускать модульные тесты с помощью встроенного в Visual Studio обозревателя тестов, доступного из меню **Test**. Если вы используете VS Code, то средство запуска тестов здесь также имеется, но необходимо настроить задачи сборки и тестирования в файле VS Code tasks.json, что позволяет запускать все тесты посредством команды **Task** > **Test**.

1.6.2 Создание собственного консольного приложения .NET Core с помощью EF Core

Я знаю, что многим разработчикам нравится создавать собственные приложения, потому что написание самого кода означает, что вы точно знаете, в чем дело. В этом разделе показано, как создать консольное приложение .NET MyFirstEfCoreApp с помощью Visual Studio.

СОЗДАНИЕ КОНСОЛЬНОГО ПРИЛОЖЕНИЯ .NET CORE

В Visual Studio есть отличный набор руководств, а на странице http://mng.bz/e56z можно найти пример создания консольного приложения C#.

> **СОВЕТ** Чтобы узнать, какую версию .NET использует ваше приложение, выберите **Project** > **MyFirstEfCoreApp Properties** в главном меню; на вкладке **Application** (Приложение) показана целевая платформа. Некоторые версии EF Core требуют определенной версии .NET Core.

ДОБАВЛЯЕМ БИБЛИОТЕКУ EF CORE В ПРИЛОЖЕНИЕ

Установить библиотеку NuGet можно различными способами. Более наглядный – использовать диспетчер пакетов NuGet; руководство

можно найти на странице http://mng.bz/pVeG. Для этого приложения вам понадобится пакет EF Core для той базы данных, к которой приложение будет подключаться. В данном случае выбираем NuGet пакет Microsoft.EntityFrameworkCore.SqlServer, потому что приложение будет использовать SQL Server для разработки, который был установлен при установке Visual Studio.

Также нужно обратить внимание на номер версии пакета NuGet, который вы собираетесь установить. EF Core устроен таким образом, что у каждого основного выпуска есть свой номер. Например, номер версии 5.1.3 означает основную версию (Major) 5, с второстепенным выпуском (Minor) 1 и патчем (исправлением ошибок – Patch) версии 3. Часто в разных проектах нужно загружать разные пакеты EF Core. Например, может потребоваться загрузить Microsoft.EntityFrameworkCore в слой доступа к данным и Microsoft.EntityFrameworkCore.SqlServer в веб-приложение. Если нужно это делать, старайтесь использовать пакеты NuGet с одинаковыми версиями Major.Minor. Patch. Если полного совпадения добиться невозможно, следите, чтобы совпадали хотя бы версии Major.Minor.

Скачивание и запуск примера приложения из репозитория Git

Есть два варианта скачивания и запуска консольного приложения MyFirstEfCoreApp из репозитория Git: Visual Studio и VS Code. Можно найти еще одно руководство по Visual Studio, «Открыть проект из репозитория», на странице http://mng.bz/OE0n. Репозиторий для этой книги находится на странице http://mng.bz/XdlG.

Обязательно выберите правильную ветку. В репозитории Git есть ветки, позволяющие переключаться между разными версиями кода. Для этой книги я создал три основные ветки: master, которая содержит код из первой части (главы 1–6); Part2, содержащую код из второй части (главы 7–11); и Part3, которая содержит код из третьей части (главы 12–17).

По умолчанию репозиторий будет открыт в ветке master, чтобы те, кто не привыкли к Git, могли сразу приступить к работе. Файл Readme в каждой ветке содержит дополнительную информацию о том, что нужно установить и что можно запустить.

1.7 База данных, к которой будет обращаться MyFirstEfCoreApp

EF Core предназначен для доступа к базе данных, но откуда берется эта база? EF Core предоставляет вам два варианта: EF Core может создать ее за вас, используя подход «*Сначала код*» (*code-first*), или вы можете предоставить существующую базу данных, созданную вами за пределами EF Core. Этот подход называется «*Сначала база данных*»

(*database-first*). В первой части книги используется первый вариант, потому что этот подход применяется многими разработчиками.

> **EF6** В EF6 можно использовать EDMX / конструктор базы данных для визуального проектирования своей базы данных. Этот вариант известен как «*Сначала проектирование*» (*design-first*). EF Core не поддерживает данный подход в какой бы то ни было форме, и планов его добавлять нет.

В этой главе мы не будем изучать, как создается база данных. Чтобы приложение MyFirstEfCoreApp работало, код самостоятельно создаст базу данных и добавит тестовые данные, если базы данных нет.

> **ПРИМЕЧАНИЕ** В своем коде я использую базовую команду EF Core, предназначенную для модульного тестирования, чтобы создать базу данных, потому что это просто и быстро. В главе 5 рассказывается, как правильно создать базу данных с EF Core, а в главе 9 полностью описана проблема создания и изменения структуры базы данных, известной как *схема* базы данных.

Для приложения MyFirstEfCoreApp я создал простую базу данных, показанную на рис. 1.2, всего с двумя таблицами:

- таблицей Books с информацией о книгах;
- таблицей Author с авторами.

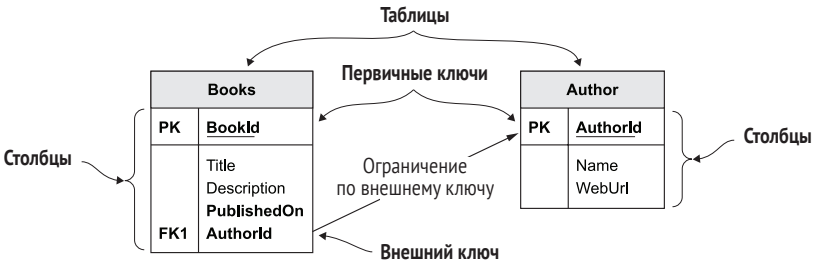
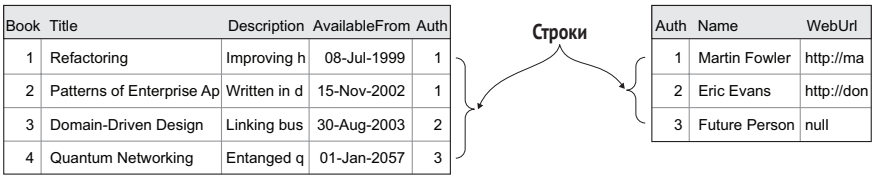


Рис. 1.2 Пример нашей реляционной базы данных с двумя таблицами: Books и Author

> **РАСШИРЕННОЕ ПРИМЕЧАНИЕ** В этом примере я позволил EF Core дать таблицам имена, используя параметры конфигурации по умолчанию. Имя таблицы Books взято из свойства `DbSet<Book> Books`, показанного на рис. 1.5. Для имени таблицы Author на рис. 1.5 нет свойства `DbSet<T>`, поэтому EF Core использует имя класса.

На рис. 1.3 показано содержимое базы данных. В ней всего четыре книги, у первых двух один и тот же автор: Мартин Фаулер.



Book	Title	Description	AvailableFrom	Auth
1	Refactoring	Improving h	08-Jul-1999	1
2	Patterns of Enterprise Ap	Written in d	15-Nov-2002	1
3	Domain-Driven Design	Linking bus	30-Aug-2003	2
4	Quantum Networking	Entanged q	01-Jan-2057	3

Auth	Name	WebUrl
1	Martin Fowler	http://ma
2	Eric Evans	http://don
3	Future Person	null

Рис. 1.3 Содержимое базы данных, где показано четыре книги, у двух из которых один и тот же автор

1.8 Настройка приложения MyFirstEfCoreApp

После создания и настройки консольного приложения .NET можно приступить к написанию кода EF Core. Перед созданием кода доступа к любой базе данных нужно написать две основные части:

- классы, которые EF Core должен отображать в таблицы вашей базы данных;
- DbContext – это основной класс, который вы будете использовать для настройки базы данных и получения доступа к ней.

1.8.1 Классы, которые отображаются в базу данных: Book и Author

EF Core отображает классы в таблицы базы данных. Следовательно, нужно создать класс, который будет определять таблицу базы данных или согласуется с ней, если база данных у вас уже есть. Существует множество правил и конфигураций (они описаны в главах 7 и 8), но на рис. 1.4 показан типичный формат класса, отображаемого в таблицу базы данных.

В листинге 1.1 показан другой класс, который вы будете использовать: `Author`. У него та же структура, что и у класса `Book` на рис. 1.4, с первичным ключом, что соответствует соглашению об именовании `<ClassName>Id` (см. раздел 7.3.5). У класса `Book` также есть навигационное свойство типа `Author` и свойство типа `int` с именем `AuthorId`, соответствующее первичному ключу `Author`. Эти два свойства сообщают EF Core, что вам нужна связь класса `Book` с классом `Author` и что свойство `AuthorId` нужно использовать как внешний ключ для связи двух таблиц в базе данных.

Листинг 1.1 Класс Author из приложения MyFirstEfCoreApp

```
public class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public string WebUrl { get; set; }
}
```

Содержит первичный ключ строки Author в БД. Обратите внимание, что у внешнего ключа из класса Book то же имя

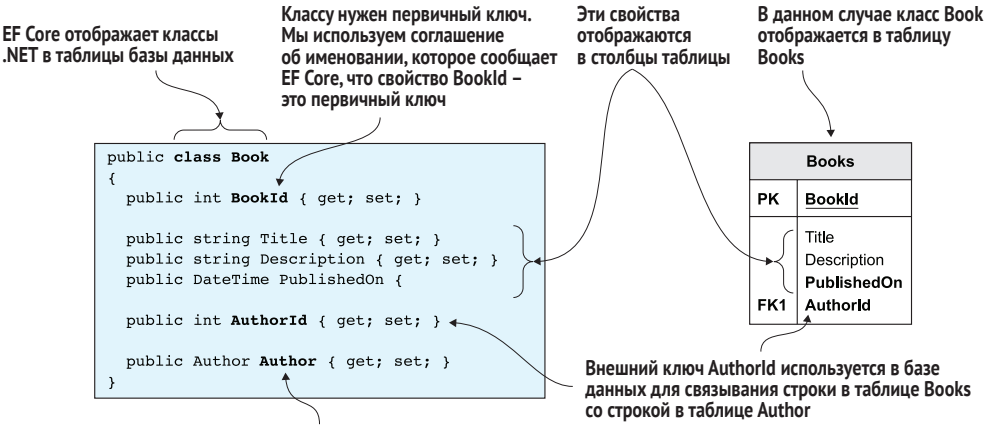


Рис. 1.4 Класс .NET Book слева отображается в таблицу базы данных Books справа. Это типичный способ создания приложения с несколькими классами, которые отображаются в таблицы базы данных

1.8.2 DbContext

Другая важная часть приложения – это DbContext, создаваемый вами класс, который наследует от класса EF Core DbContext. Этот класс содержит необходимую информацию, чтобы настроить отображение в базу данных. Этот класс используется в коде для доступа к базе данных (см. раздел 1.9.2). На рис. 1.5 показан класс DbContext, `AppDbContext`, используемый консольным приложением MyFirstEfCoreApp.

В нашем небольшом приложении все решения по моделированию принимает EF Core, используя набор соглашений. Есть много различных способов сообщить EF Core, что такое модель базы данных, и эти команды могут быть сложными. Потребуется материал из глав 7 и 8 и немного сведений из главы 10, чтобы охватить все варианты, доступные вам как разработчику.

Кроме того, используется стандартный подход для настройки доступа к базе данных в консольном приложении: переопределение метода `OnConfiguring` внутри DbContext и предоставление всей информации, необходимой EF Core для определения типа и местоположения базы данных. Недостатком такого подхода является то, что имеется фиксированная строка подключения, и это затрудняет разработку и модульное тестирование.

Для веб-приложений ASP.NET Core эта проблема серьезнее, потому что одновременно нужен доступ к локальной базе данных для тестирования и к базе данных, развернутой в промышленном окружении. В главе 2, когда вы начнете создавать веб-приложение ASP.NET Core, вы познакомитесь с другим подходом, который позволит изменять строку подключения к базе данных (см. раздел 2.2.2).

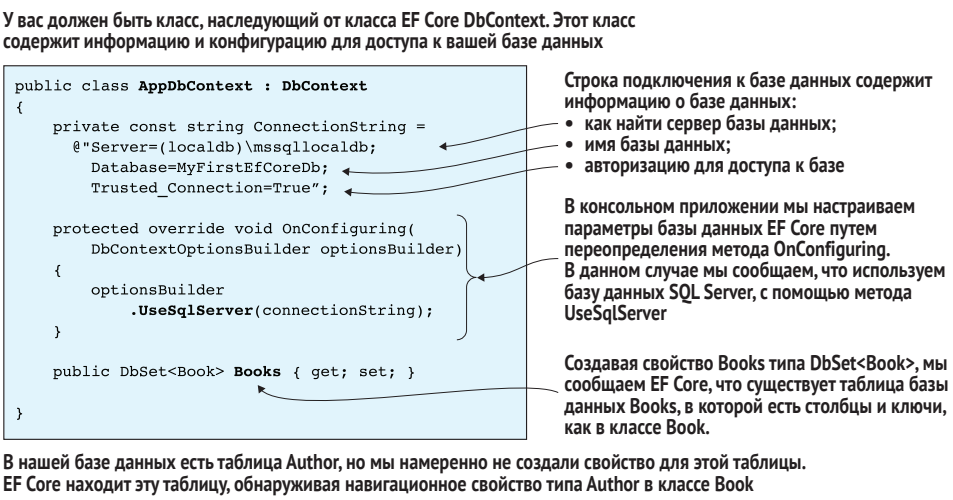


Рис. 1.5 Две основные части DbContext, созданные для консольного приложения MyFirstEfCoreApp. Сначала настройка параметров базы данных определяет, какой тип базы данных использовать и где ее можно найти. Далее свойство (или свойства) `DbSet<T>` сообщает EF Core, какие классы нужно отобразить в базу данных

1.9 Заглянем под капот EF Core

Теперь, когда мы запустили приложение MyFirstEfCoreApp, стоит разобраться, как работает библиотека EF Core. Акцент делается не на коде приложения, а на том, что происходит внутри библиотеки при чтении и записи данных в базу данных. Моя цель – предоставить вам ментальную модель того, как EF Core обращается к базе данных. Эта модель должна помочь вам, когда вы углубитесь в бесчисленное множество команд, описанных в оставшейся части книги.

Действительно ли нужно знать внутреннее устройство EF Core, чтобы использовать его?

Можно использовать библиотеку EF Core, не беспокоясь о том, как он работает. Но знание того, что происходит внутри, поможет понять, почему различные команды работают именно так. Кроме того, вы будете лучше вооружены, когда нужно будет отладить код доступа к базе данных.

Следующие страницы содержат множество пояснений и диаграмм, чтобы показать вам, что происходит внутри EF Core. EF Core «скрывает» базу данных, чтобы вы как разработчик могли легко писать код доступа к базе данных, что хорошо работает на практике. Но, как я уже сказал, знание того, как работает EF Core, может помочь вам, если вы хотите сделать что-то посложнее или если что-то работает не так, как ожидалось.

1.9.1 Моделирование базы данных

Прежде чем вы сможете что-либо делать с базой данных, EF Core должен пройти процесс, который я называю *моделированием базы данных*. Моделирование – способ EF Core определить, как выглядит база данных, глядя на классы и другие данные конфигурации. Затем EF Core использует полученную модель при всех обращениях к базе данных.

Процесс моделирования запускается при первом создании класса DbContext, в данном случае `AppDbContext` (показанного на рис. 1.5). У него есть одно свойство `DbSet<Book>` – это способ доступа кода к базе данных.

На рис. 1.6 представлен общий вид процесса, используемого EF Core для моделирования базы данных. В последующих главах вы познакомитесь с рядом команд, которые позволят вам более точно настроить базу данных, а пока вы будете использовать конфигурации по умолчанию.

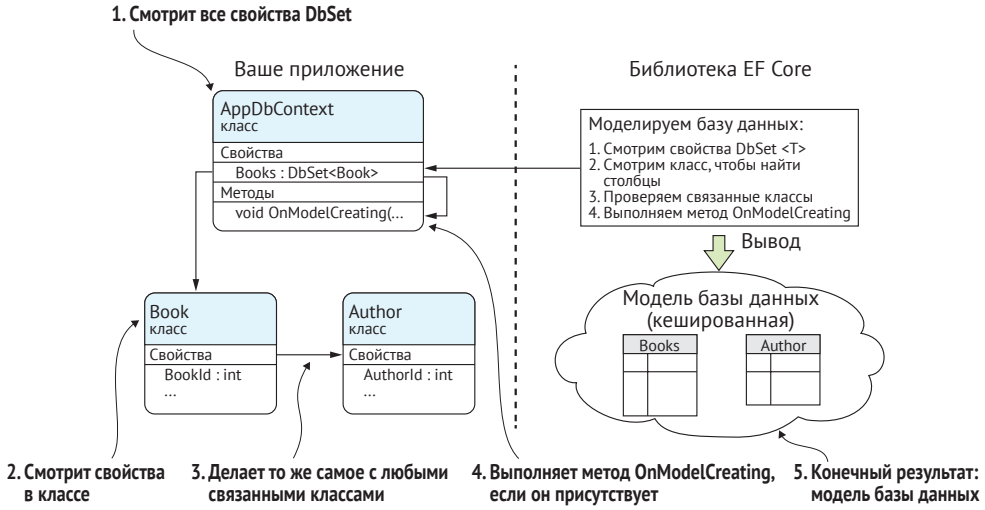


Рис. 1.6 На рисунке показано, как EF Core создает модель базы данных, в которую отображаются ваши классы. Сначала он смотрит на классы, которые вы определили с помощью свойств `DbSet<T>`, а потом смотрит на все ссылки на другие классы. Используя эти классы, EF Core может построить модель базы данных по умолчанию. Но далее он выполняет метод `OnModelCreating` в DbContext, который можно переопределить, чтобы добавить свои конкретные команды и настроить базу данных так, как вы хотите

На рис. 1.6 показаны этапы моделирования, которые EF Core использует в классе `AppDbContext`. Это происходит при первом создании экземпляра класса. (После этого модель кешируется, поэтому последующие экземпляры создаются быстро.) Ниже представлено более подробное описание процесса:

- EF Core просматривает DbContext приложения и находит все публичные свойства `DbSet<T>`. На основе этих данных он определяет начальное имя для одной найденной таблицы: Books;

- EF Core просматривает все классы, на которые ссылаются в `DbSet<T>`, и смотрит их свойства для определения имен столбцов, типов и т. д., а также ищет специальные атрибуты класса и/или свойств, обеспечивающие дополнительную информацию о моделировании;
- EF Core ищет любые классы, на которые ссылаются классы `DbSet<T>`. В нашем случае у класса `Book` есть ссылка на класс `Author`, поэтому EF Core сканирует и его. Он выполняет тот же поиск по свойствам класса `Author`, что и в случае с классом `Book` на этапе 2, а также принимает имя класса `Author` в качестве имени таблицы;
- напоследок EF Core вызывает виртуальный метод `OnModelCreating` внутри DbContext. В этом простом приложении вы не переопределяете метод `OnModelCreating`, но если бы вы это сделали, то могли бы предоставить дополнительную информацию через Fluent API, чтобы выполнить настройку моделирования;
- EF Core создает внутреннюю модель базы данных на основе собранной информации. Эта модель базы данных кешируется, поэтому последующий доступ будет быстрее. Затем она используется для выполнения всех обращений к базе данных.

Возможно, вы заметили, что на рис. 1.6 базы данных нет. Это объясняется тем, что когда EF Core создает свою внутреннюю модель, он не смотрит на базу данных. Я подчеркиваю этот факт, чтобы показать, насколько важно построить хорошую модель нужной вам базы данных; в противном случае могут возникнуть проблемы, если существует несоответствие между тем, как по мнению EF Core должна выглядеть база данных, и фактической базой.

В своем приложении вы можете использовать EF Core для создания базы данных, и в этом случае шансы на несоответствие равны нулю. Тем не менее если вам нужна хорошая и эффективная база данных, стоит позаботиться о том, чтобы создать правильное представление нужной базы данных в своем коде, чтобы созданная база работала нормально. Варианты создания, обновления и управления структурой базы данных – обширная тема, подробно описанная в главе 9.

1.9.2 Чтение данных

Теперь у вас есть доступ к базе данных. У консольного приложения есть команда `l (list)`, которая читает данные из базы и выводит информацию в терминал. На рис. 1.7 показан результат запуска консольного приложения с командой l.

В следующем листинге показан код, который вызывается для вывода списка всех книг с авторами в консоль.

Листинг 1.2 Код для перечисления всех книг и вывода их в консоль

```
public static void ListAll()
{
    using (var db = new AppDbContext())
```

Создается DbContext приложения, через который выполняются все обращения к базе данных

```
    {
        foreach (var book in
            db.Books.AsNoTracking()
            .Include(book => book.Author))
        {
            var webUrl = book.Author.WebUrl == null
                ? "- no web URL given -"
                : book.Author.WebUrl;
            Console.WriteLine(
                $"{book.Title} by {book.Author.Name}");
            Console.WriteLine(" " +
                "Published on " +
                $"{book.PublishedOn:dd-MMM-yyyy}" +
                $". {webUrl}");
        }
    }
}
```

Читаются данные обо всех книгах. AsNoTracking указывает на то, что доступ к данным осуществляется в режиме «только для чтения»

Метод Include делает так, чтобы с каждой книгой загружалась информация об авторе. Смотрите главу 2 для получения дополнительной информации

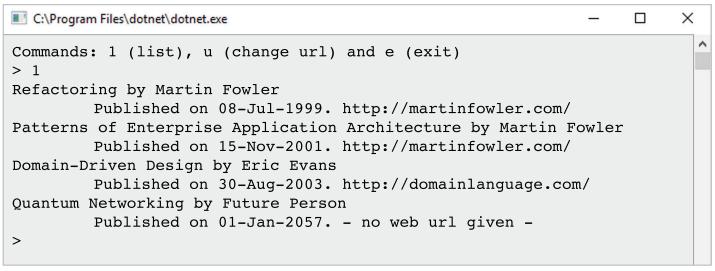


Рис. 1.7 Вывод консольного приложения при перечислении содержимого базы данных

EF Core использует язык Language Integrated Query (LINQ) от Microsoft для передачи нужных команд и обычные классы .NET для хранения данных. Запрос из листинга 1.2 не включает никаких методов LINQ, но позже вы увидите много примеров с LINQ.

ПРИМЕЧАНИЕ *Изучение LINQ будет иметь важное значение для вас, поскольку EF Core использует команды LINQ для доступа к базе данных.* В приложении содержится краткое введение в LINQ. Также доступно множество онлайн-ресурсов; см. http://mng.bz/YqBN.

Две строки кода, выделенные жирным шрифтом в листинге 1.2, обеспечивают доступ к базе данных. Теперь посмотрим, как EF Core использует этот код для доступа к базе данных и возврата необходимых книг с указанием их авторов. На рис. 1.8 показано, как выполняется запрос к базе данных.

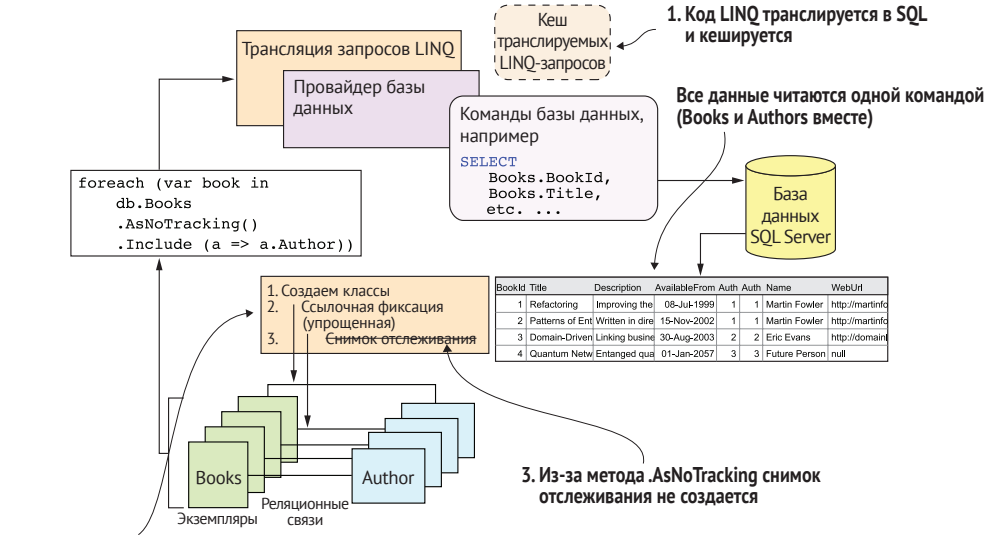


Рис. 1.8 Как EF Core выполняет запрос к базе данных

Процесс чтения данных из базы данных выглядит следующим образом:

- запрос `db.Books.AsNoTracking().Include(book => book.Author)` обращается к свойству `DbSet <Book>` в DbContext приложения и добавляет `.Include(book => book.Author)` в конце, чтобы запросить загрузку связанных данных из `Author`. Далее провайдер базы данных выполняет преобразование в SQL-команду для доступа к базе данных. Полученный в результате SQL запрос кешируется, чтобы избежать затрат на повторную трансляцию, если тот же запрос к базе данных будет использоваться снова.

EF Core пытается максимально эффективно использовать доступ к базе данных. В данном случае он объединяет две таблицы, которые ему нужно прочитать, Books и Author, в одну большую таблицу, чтобы можно было выполнять работу за один запрос к базе данных. В следующем листинге показан SQL запрос, созданный EF Core и провайдером базы данных.

Листинг 1.3 Команда SQL, созданная для чтения данных из таблиц Books и Author

```
SELECT [b].[BookId],
[b].[AuthorId],
[b].[Description],
[b].[PublishedOn],
```

```
[b].[Title],
[a].[AuthorId],
[a].[Name],
[a].[WebUrl]
FROM [Books] AS [b]
INNER JOIN [Author] AS [a] ON
[b].[AuthorId] = [a].[AuthorId]
```

После того как провайдер базы данных прочитает данные, EF Core передает их, используя процесс, который (а) создает экземпляры классов .NET и (б) использует ссылки реляционной базы данных, которые называются *внешними ключами*, для правильного связывания классов .NET по ссылке. Это называется ссылочная фиксация (relational fixup). Поскольку мы добавили метод `AsNoTracking`, используется упрощенная фиксация по соображениям скорости.

ПРИМЕЧАНИЕ Различия между упрощенной фиксацией с использованием метода `AsNoTracking` и обычной ссылочной фиксацией обсуждаются в разделе 6.1.2.

Результат – набор экземпляров классов .NET со свойством `Author` класса `Book`, связанным с классом `Author`, содержащим информацию об авторе. В этом примере у двух книг один автор, Мартин Фаулер, поэтому есть два экземпляра класса `Author`, каждый из которых содержит одну и ту же информацию о Мартине Фаулере.

Поскольку этот код включает команду `AsNoTracking`, EF Core знает, как подавить создание *снимка отслеживания изменений данных* (*tracking snapshot*). Снимки отслеживания используются для обнаружения изменений данных, как будет показано в примере редактирования столбца базы данных WebUrl в разделе 1.9.3. Поскольку этот запрос с доступом только на чтение, отключение моментального снимка отслеживания делает команду быстрее.

1.9.3 Обновление

Теперь нужно использовать вторую команду update (`u`) в MyFirstEfCoreApp, чтобы обновить столбец WebUrl в таблице Author книги «*Quantum Networking*». Как показано на рис. 1.9, сначала перечисляются все книги, чтобы показать, что у последней книги нет URL-адреса автора. Затем выполняется команда `u`, которая запрашивает новый URL-адрес автора для последней книги, «*Quantum Networking*». Вы вводите новый URL-адрес `httqs://entangled.moon` (это книга о вымышленном будущем, так почему бы не использовать вымышленный URL-адрес!), и после обновления команда снова перечисляет все книги, показывая, что URL-адрес автора изменился (красным обведено место, где должен быть указан URL-адрес, и адрес, который появился после).

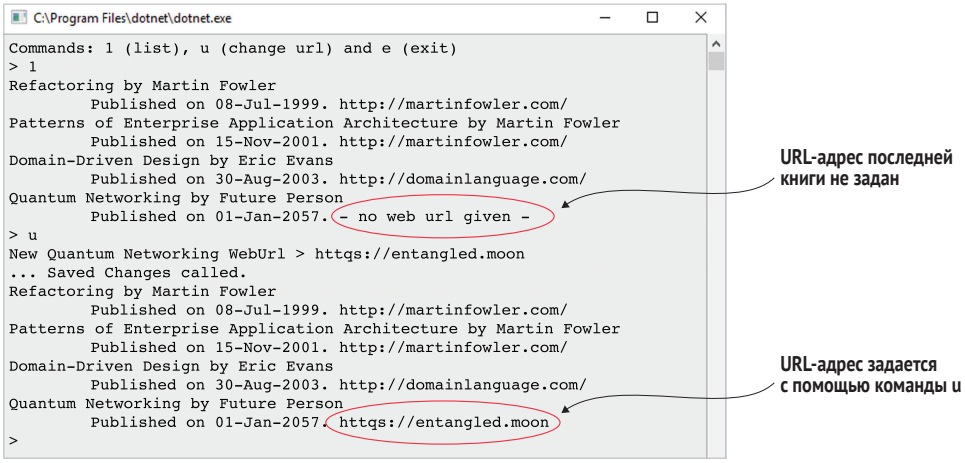


Рис. 1.9 На этом рисунке показано обновление в действии. Первая команда – l (list), которая показывает каждую книгу, имя ее автора и URL-адрес на следующей строке. Затем вы нажимаете u (update), что позволяет обновить URL-адрес автора последней книги. Команда обновления вызывает команду list, чтобы было видно, что обновление прошло успешно

Код для обновления столбца WebUrl в таблице Author, связанной с книгой «*Quantum Networking*», показан здесь:

Листинг 1.4. Код для обновления столбца WebUrl

```
public static void ChangeWebUrl()
{
    Console.Write("New Quantum Networking WebUrl > ");
    var newWebUrl = Console.ReadLine();

    using (var db = new AppDbContext())
    {
        var singleBook = db.Books
            .Include(book => book.Author)
            .Single(book => book.Title == "Quantum Networking");

        singleBook.Author.WebUrl = newWebUrl;
        db.SaveChanges();
        Console.WriteLine("... SavedChanges called.");
    }
    ListAll();
}
```

Считывается из консоли новый URL-адрес

Загружается информация об авторе вместе с книгой

Выбирается книга с названием Quantum Networking

Чтобы внести изменения в базу данных, изменяются значения, которые были прочитаны

Метод SaveChanges сообщает EF Core о необходимости проверки всех изменений считанных данных и записи этих изменений в базу

Отображается вся информация о книгах

На рис. 1.10 показано, что происходит внутри библиотеки EF Core. Этот пример намного сложнее, чем предыдущий пример чтения данных, поэтому позвольте дать вам несколько советов по поводу того, что искать.

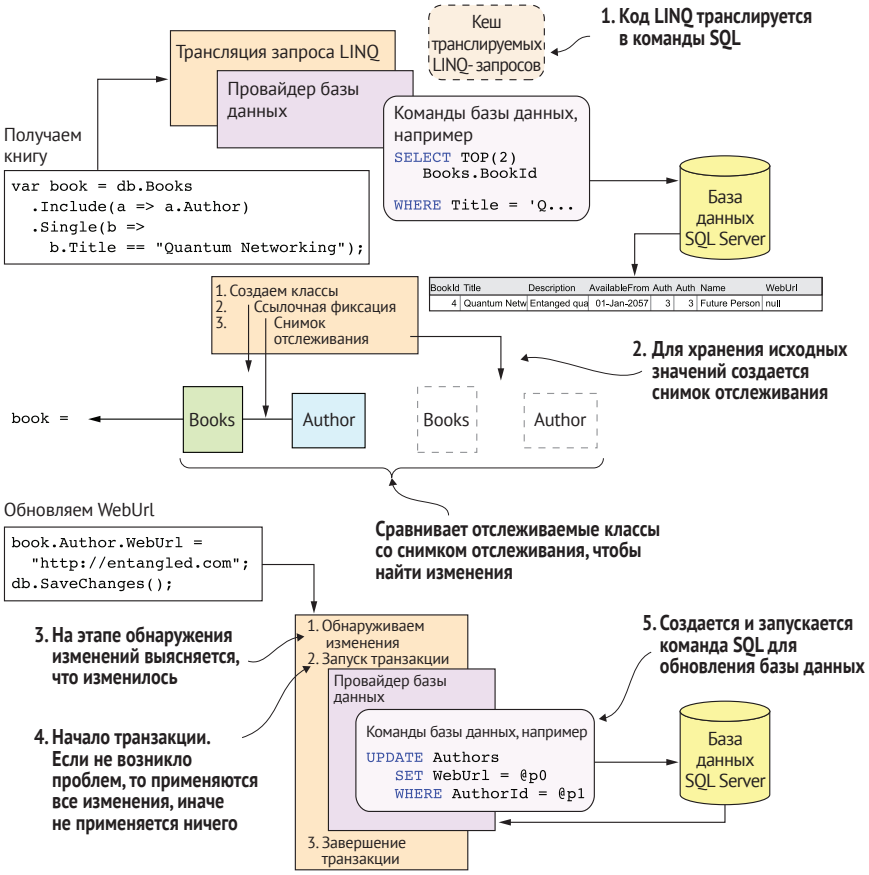


Рис. 1.10 На этом рисунке показано, что делает EF Core, когда вы обновляете свойство `WebUrl` и просите EF Core записать его в базу данных. Это довольно сложный рисунок, но если начать сверху и следовать за пронумерованным текстом, понять его будет легче. Все начинается с чтения, чтобы получить нужную книгу и автора. (Обратите внимание, что в этом процессе присутствует снимок отслеживания; см. этап 2.) Затем, когда вы обновляете `WebUrl` и вызываете метод `SaveChanges`, EF Core создает и выполняет правильную SQL-команду для обновления столбца WebUrl в правильной строке

Во-первых, этап чтения в верхней части диаграммы аналогичен примеру чтения и поэтому должен быть вам знаком. В данном случае запрос загружает конкретную книгу, используя ее заголовок в качестве фильтра. Важным изменением является пункт 2: создается снимок отслеживания.

Изменение происходит на этапе обновления в нижней половине диаграммы. Здесь видно, что EF Core сравнивает загруженные данные со снимком отслеживания, чтобы обнаружить изменения. Он видит, что было изменено только свойство `WebUrl`, и создает команду SQL, чтобы обновить только столбец WebUrl в нужной строке таблицы Author.

Я описал большинство шагов, а вот подробное описание того, как обновляется столбец WebUrl:

1. Приложение использует LINQ-запрос для поиска конкретной книги с информацией об авторе. EF Core превращает запрос в команду SQL для чтения строк, где Title имеет значение *Quantum Networking*, возвращая экземпляры обоих классов `Book` и `Author`, и проверяет, что была найдена только одна строка.
2. LINQ-запрос не включает метод `.AsNoTracking`, который использовался в предыдущих версиях чтения данных, поэтому запрос считается *отслеживаемым*. Следовательно, EF Core создает снимок загруженных данных.
3. Затем код изменяет свойство `WebUrl` в классе книги `Author`. При вызове метода `SaveChanges` на этапе обнаружения изменений сравниваются все классы, которые были возвращены из отслеживаемого запроса со снимком отслеживания. Так можно определить, что изменилось – в данном случае только свойство `WebUrl` класса `Author` с первичным ключом 3.
4. При обнаружении изменения EF Core начинает *транзакцию*. Каждое обновление базы данных выполняется как *атомарная единица*: если в базе данных происходит несколько изменений, они либо все успешные, либо все терпят неудачу. Это важный факт, потому что реляционная база данных может перейти в ненадлежащее состояние, если будет применена только часть изменений.
5. Запрос на обновление преобразуется провайдером базы данных в SQL-команду, которая выполняет обновление. Если команда была выполнена успешно, транзакция фиксируется, и происходит выход из метода `SaveChanges`; в противном случае возникает исключение.

1.10 Этапы разработки EF Core

EF Core и .NET Core прошли долгий путь с момента первого выпуска. Корпорация Microsoft усердно работала над улучшением производительности .NET Core, при этом добавляя дополнительные функциональные возможности до такой степени, что .NET 5 может заменить существующую платформу .NET Framework 4.8.

На рис. 1.11 показана история основных выпусков EF Core до настоящего момента. Номера версий EF Core следуют за номером версии NET Core. Обратите внимание, что выпуски в верхней части рисунка – это выпуски *с долгосрочной поддержкой* (LTS). Это означает, что выпуск поддерживается в течение трех лет после первого выпуска. Основные выпуски ожидаются каждый год, а LTS-релизы – каждые два года.

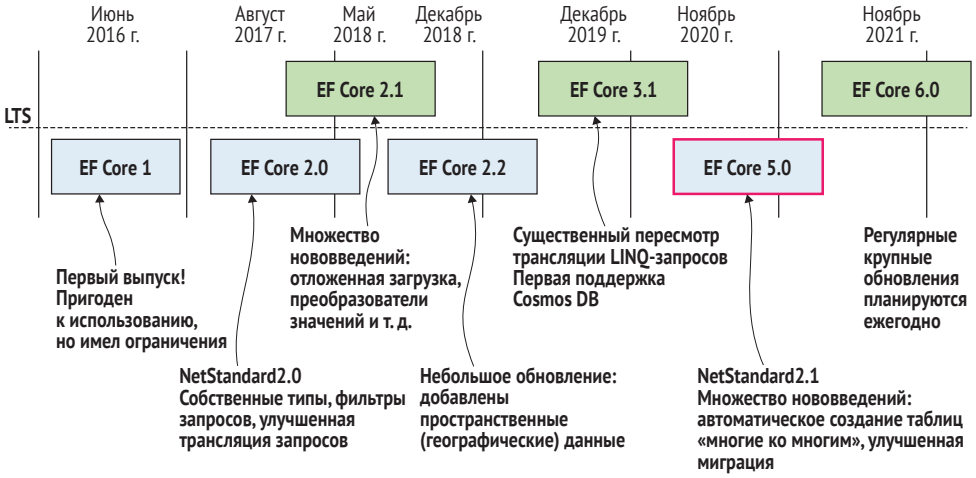


Рис. 1.11 На этом рисунке показана разработка EF Core, которая идет параллельно с разработкой NET, платформы для разработчиков с открытым исходным кодом. Версия EF Core 5 выделена, потому что эта книга охватывает все функциональные возможности EF Core до EF Core 5 включительно

1.11 Стоит ли использовать EF Core в своем следующем проекте?

Теперь, когда у вас есть краткий обзор того, что такое EF Core и как он работает, встает вопрос: стоит ли начинать использовать EF Core в своем проекте? Для всех, кто планирует перейти на EF Core, ключевой вопрос заключается в следующем: «Достаточно ли EF Core превосходит библиотеку доступа к данным, которую я использую сейчас, чтобы использовать ее в моем следующем проекте?» Изучение и внедрение любой новой библиотеки, особенно такой сложной, как EF Core, связано с затратами, поэтому это правильный вопрос. Вот мой взгляд на EF Core и .NET Core в целом.

1.11.1 .NET – это программная платформа будущего, и она будет быстрой!

Microsoft давно и упорно работала над улучшением производительности .NET Core, при этом добавляя дополнительные функциональные возможности. Ориентация на производительность перебросила веб-приложения на Microsoft ASP.NET Core с примерно 250-го места (ASP.NET MVC) на примерно 10–40-е места (ASP.NET Core) (в зависимости от нагрузки); см. http://mng.bz/Gxaq. Аналогичный, но меньший прирост производительности был добавлен в EF Core.

Microsoft заявила, что .NET 5 заменит существующую платформу .NET Framework 4.8, однако вспышка COVID-19 сорвала эти планы, и теперь .NET 6 заменит .NET Framework 4.8. Но здесь есть ясный сигнал: если вы начинаете новый проект, а у .NET 5 и EF Core есть функциональные возможности, необходимые вашему проекту, то переход на EF Core означает, что вы не останетесь за бортом.

1.11.2 Открытый исходный код и открытые сообщения

За многие годы Microsoft изменилась. Вся ее работа над .NET Core – это открытый исходный код, и множество внешних специалистов занимаются исправлением ошибок и добавлением нововведений, поэтому у вас может быть прямой доступ к коду, если он вам нужен.

Кроме того, впечатляет уровень открытости сообщений о том, что происходит в .NET Core и других продуктах. Команда EF Core, например, еженедельно выпускает обновления о том, что она делает, предоставляя множество ранних предварительных просмотров новых выпусков и делая регулярные ночные сборки EF Core доступными для всех. Команда серьезно относится к отзывам, а все работы и дефекты отображаются на страницах проблем репозитория EF Core.

1.11.3 Мультиплатформенные приложения и разработка

Как я сказал в начале главы, EF Core поддерживает несколько платформ; вы можете выполнять разработку и запускать приложения EF Core в Windows, Linux и Apple. Этот факт означает, что можно запускать приложения Microsoft на дешевых системах Linux. Также вполне возможна разработка на разных платформах. Артур Викерс, который является одним из ведущих инженеров команды EF Core, решил перейти с Windows на Linux в качестве основной платформы разработки. О его опыте можно прочитать на странице http://mng.bz/zxWa.

1.11.4 Быстрая разработка и хорошие функциональные возможности

Моя основная работа – разработчик по контракту. В типичном приложении, управляемом данными, я пишу большое количество кода доступа к базе данных, причем некоторые его части сложны. С EF Core я могу писать код доступа очень быстро и сделать это так, чтобы он был легким для понимания и рефакторинга, если он слишком медленный. Это основная причина, по которой я использую EF Core.

В то же время мне нужен инструмент объектно-реляционного отображения со множеством функциональных возможностей, чтобы я мог создать базу данных так, как я хочу, не сталкиваясь со слишком большим количеством препятствий в EF Core. Конечно, некоторые вещи, такие как создание общих табличных выражений SQL, исклю-

чены, но использование чистого SQL в некоторых случаях позволяет обходить подобные ограничения, если мне это нужно.

1.11.5 Хорошая поддержка

По EF Core есть хорошая документация (https://docs.microsoft.com/en-us/ef/core/index), и, конечно же, у вас есть эта книга, в которой собрана документация с подробными объяснениями и примерами, а также паттерны и практики, которые сделают вас отличным разработчиком. В интернете полно блогов об EF Core, в том числе и мой на странице https://www.thereformedprogrammer.net. А для вопросов и ошибок всегда есть Stack Overflow; http://mng.bz/0mDx.

Другая часть поддержки – это инструменты разработки. Microsoft, кажется, изменила фокус, предоставив поддержку нескольких платформ, но еще создала бесплатную кросс-платформенную среду разработки под названием VS Code, а также сделала свой основной инструмент разработки, Visual Studio (Windows и Mac), бесплатным для индивидуальных разработчиков и предприятий малого бизнеса; в разделе «Использование» в нижней части страницы по адресу www.visualstudio.com/vs/community подробно описаны все условия. Привлекательное предложение.

1.11.6 Всегда высокая производительность

Ах да, проблема с производительностью базы данных. Послушайте, я не буду говорить, что EF Core сразу же обеспечит невероятную производительность доступа к базе данных с красивым SQL и быстрым получением, внесением и обработкой данных для последующего их использования или хранения в базе. Это цена, которую вы платите за быструю разработку кода доступа к данным; вся эта «магия» внутри EF Core не может сравниться с написанным вручную SQL, но, возможно, вы удивитесь, насколько эффективной она может быть. Смотрите главу 15, где я по шагам настраиваю производительность приложения.

Но есть много возможностей улучшить производительность своих приложений. В моих приложениях я обнаружил, что только около 5–10 % запросов являются ключевыми, которым нужна ручная настройка. Главы 14 и 15 посвящены настройке производительности, как и часть главы 16. Эти главы показывают, что можно многое сделать для повышения производительности доступа к базе данных с помощью EF Core.

Также нет причин, по которым нельзя перейти на чистый SQL для доступа к базе данных. Это замечательно: быстро создать приложение с помощью EF Core, а затем преобразовать несколько мест, где EF Core не обеспечивает хорошую производительность, в команды на чистом SQL, используя ADO.NET или Dapper.

1.12 Когда не следует использовать EF Core?

Я явный сторонник EF Core, но не буду использовать его в клиентском проекте, если это не имеет смысла. Итак, рассмотрим несколько моментов, когда можно решить *не* использовать EF Core.

Первый очевиден: поддерживает ли он базу данных, которую вы хотите использовать? Список поддерживаемых баз данных можно найти на странице https://docs.microsoft.com/en-us/ef/core/providers.

Второй фактор – это необходимый уровень производительности. Если вы пишете, скажем, небольшой REST-совместимый сервис или бессерверную систему, то я не уверен, что EF Core – подходящий вариант; вы можете использовать быструю, но требовательную ко времени разработки библиотеку, потому что у вас не так много обращений к базе данных. Но если у вас большое приложение, со множеством скучных административных API и важными API для клиентов, вам может подойти гибридный подход. (Смотрите главу 15, где приводится пример приложения, написанного с использованием EF Core и Dapper.)

Кроме того, EF Core не очень подходит для пакетных (bulk) команд. Обычно такие задачи, как массовая загрузка больших объемов данных и удаление всех строк в таблице, можно реализовать быстрее, используя чистый SQL. Но несколько расширений EF Core для пакетных CRUD-операций (некоторые с открытым исходным кодом, а некоторые платные) могут помочь; попробуйте выполнить поиск по запросу «пакетная загрузка в EF Core» («EF Core bulk loading»), чтобы найти возможные библиотеки.

Резюме

- EF Core – инструмент объектно-реляционного отображения (ORM), использующий язык LINQ для определения запросов к базе данных и возврата данных в связанные экземпляры классов .NET.
- EF Core разработан для быстрого и интуитивно понятного написания кода для доступа к базе данных. У нее есть множество функциональных возможностей, соответствующих многим требованиям.
- Были показаны различные примеры того, что происходит внутри EF Core. Эти примеры помогут вам понять, что могут делать команды EF Core, описанные в последующих главах.
- Есть много веских причин рассмотреть возможность использования EF Core: за ним стоит большой опыт, у него хорошая поддержка, и он работает на нескольких платформах.

Для читателей, знакомых с EF6.x:

- ищите в книге примечания по EF6. В них отмечены различия между подходом на базе EF Core и подходом на базе EF6.x. Также не за-

бывайте про резюме в конце каждой главы, где указаны основные изменения EF Core в этой главе;

- рассматривайте EF Core как новую библиотеку, которая была написана для имитации EF6.x, но работает по-другому. Такой образ мышления поможет вам определить улучшения EF Core, меняющие способ доступа к базе данных;
- EF Core больше не поддерживает подход EDMX / конструктор баз данных, который использовался в более ранних версиях EF.

2 Выполнение запроса к базе данных

В этой главе рассматриваются следующие темы:

- моделирование трех основных типов связей в базе данных;
- создание и изменение базы данных посредством миграции;
- определение и создание DbContext;
- загрузка связанных данных;
- разделение сложных запросов на подзапросы.

Эта глава посвящена использованию EF Core для чтения данных, т. е. выполнению запросов к базе данных. Вы создадите базу данных, содержащую три основных типа связей, которые встречаются в EF Core. Попутно вы научитесь создавать и изменять структуру базы данных с помощью EF Core.

Затем вы узнаете, как получить доступ к базе данных с помощью EF Core, считывая данные из таблиц. Вы изучите основной формат запросов EF Core, прежде чем рассматривать различные подходы к загрузке связанных и основных данных, например загрузке данных об авторе вместе с информацией о книге из главы 1.

Изучив способы загрузки связанных данных, вы начнете создавать более сложные запросы, необходимые для работы сайта по продаже книг. Эта задача охватывает сортировку, фильтрацию и разбиение на страницы, а также подходы, сочетающие эти отдельные команды для создания одного составного запроса к базе данных.

СОВЕТ Я использую модульные тесты (unit tests), чтобы убедиться, что то, что я пишу в этой книге, является правильным. Вы можете изучить и запустить эти тесты – они могут помочь вам понять происходящее. Их можно найти в соответствующем репозитории GitHub на странице http://mng.bz/XdlG. Просмотрите файл Readme из репозитория для получения информации о том, где найти модульные тесты и как их запустить.

2.1 Закладываем основу: наш сайт по продаже книг

В этой главе мы начнем создавать сайт по продаже книг, который с этого момента будет называться *Book App*. Данное приложение представляет собой хорошее средство, чтобы увидеть, что такое связи. В этом разделе представлена база данных, различные классы и части EF Core, необходимые приложению Book App для доступа к базе данных.

2.1.1 Реляционная база данных приложения Book App

Хотя мы могли бы создать базу данных, содержащую все данные о книге, ее авторе (авторах) и отзывах в одной таблице, в случае с реляционной базой данных это работало бы не очень хорошо, особенно потому, что отзывы различаются по длине. Норма для реляционных баз данных – разделение любых повторяющихся данных (например, авторов).

Расположить различные части информации о книге в базе данных можно несколькими способами, но в этом примере разместим данные так, чтобы в базе данных было по одному из основных типов связей, которые можно встретить в EF Core. Вот эти связи:

- «один к одному» (one-to-one) – PriceOffer и Book;
- «один ко многим» (one-to-many) – Book и Reviews;
- «многие ко многим» (many-to-many) – Books, связанные с Authors, и Books, связанные с Tags.

СВЯЗЬ «ОДИН К ОДНОМУ»: PRICEOFFER И BOOK

У книги может быть акционная цена с дополнительной строкой в PriceOffer. Это пример связи «один к одному». (Технически это связь «один к нулю или единице», но для EF Core это одно и то же.) Смотрите рис. 2.1.

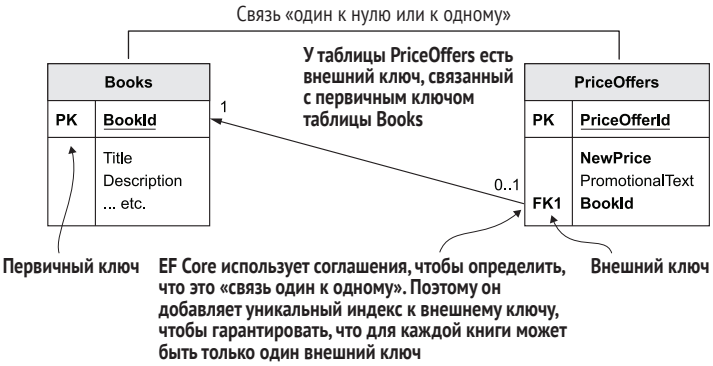


Рис. 2.1 Связь «один к одному» между Book и необязательным PriceOffer. Если PriceOffer связано с Book, то NewPrice в PriceOffer имеет приоритет над Price в Book

Чтобы рассчитать окончательную стоимость книги, нужно проверить наличие строки в таблице PriceOffer, которая связана с Books посредством внешнего ключа. Если такая строка найдена, NewPrice заменяет цену исходной книги, и на экране отображается рекламный текст (PromotionalText), как в этом примере:

~~$40~~ $30 Специальная летняя цена, только на этой неделе!

ДОПОЛНИТЕЛЬНЫЕ ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ

В этом примере у меня есть первичный (primary key) и внешний (foreign key) ключи, чтобы легче было понять связи. Но в случае связи «один к одному» также можно сделать внешний ключ первичным. В таблице PriceOffer, показанной на рис. 2.1, у вас будет первичный ключ BookId, который также будет внешним ключом. В результате вы теряете столбец PriceOfferId, что делает таблицу несколько более эффективной со стороны базы данных. Мы обсудим эту тему позже, в разделе 8.6.1.

СВЯЗЬ «ОДИН КО МНОГИМ»: REVIEWS И BOOK

Мы хотим, чтобы покупатели могли оставить отзыв о книге: они могут присвоить ей рейтинг в звездах и при желании оставить комментарий. Поскольку на книгу может не быть отзывов или их может быть много (неограниченное количество), необходимо создать таблицу для хранения этих данных. В этом примере мы назовем таблицу Review. У таблицы Books имеется связь «один ко многим» с таблицей Review, как показано на рис. 2.2.

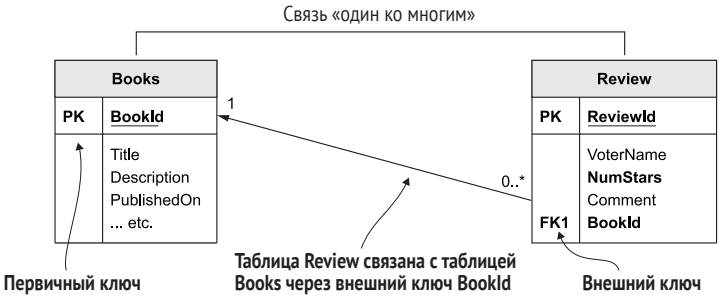


Рис. 2.2 Связь «один ко многим» между книгой и нулем или многими отзывами на нее. Эти отзывы работают так же, как и на любом сайте электронной коммерции, например Amazon

На экране «Сводка» нужно подсчитать количество отзывов и вычислить средний рейтинг, чтобы показать сводку. Вот типичная надпись на экране, которая могла бы получиться из связи «один ко многим»:

Оценки: 4,5 от 2 покупателей

СВЯЗЬ «МНОГИЕ КО МНОГИМ»: НАСТРОЙКА ВРУЧНУЮ

Книги могут быть написаны одним или несколькими авторами, а автор может написать одну или несколько книг. Следовательно, вам понадобится таблица с именем Books для хранения данных о книгах и еще одна таблица с именем Authors для хранения фамилий и имен авторов. Связь между этими таблицами называется *многие ко многим*, и в этом случае нам потребуется связующая таблица.

Мы создадим нашу собственную связующую таблицу с полем Order, потому что имена авторов в книге должны отображаться в определенном порядке (рис. 2.3).

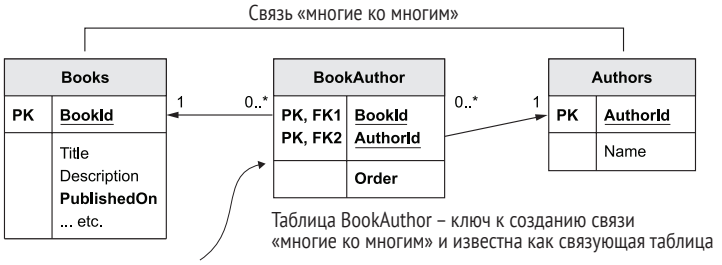


Рис. 2.3 Три таблицы, участвующие в создании связи «многие ко многим» между таблицами Books и Authors. Я использую эту связь, потому что у книги может быть много авторов, а авторы могут написать много книг. Здесь необходима дополнительная функциональная возможность – значение Order. Поскольку порядок, в котором авторы перечислены в книге, важен, я использую колонку Order, чтобы фамилии и имена авторов отображались в правильной последовательности

Вот как это будет выглядеть на экране:

Дино Эспозито, Андреа Сальтарелло

СВЯЗЬ МНОГИЕ КО МНОГИМ: АВТОМАТИЧЕСКАЯ КОНФИГУРАЦИЯ EF CORE

Книги можно пометить в зависимости от категории, например Microsoft .NET, Linux, Web и т. д., чтобы покупателю было проще найти книгу по интересующей его теме. Категорию можно применить к нескольким книгам, а в книге может быть одна или несколько категорий, поэтому необходима связующая таблица «многие ко многим». Но, в отличие от предыдущей таблицы BookAuthor, теги не нужно упорядочивать, что упрощает таблицу.

EF Core 5 и более поздние версии могут автоматически создать связующую таблицу «многие ко многим» за вас. На рис. 2.4 показана наша база данных с автоматически созданной таблицей BookTag, которая предоставляет связь «многие ко многим» между таблицами Books и Tags. Таблица BookTag выделена серым цветом, чтобы обозначить тот факт, что EF Core создает ее автоматически и что она не отображается ни в один из созданных вами классов.

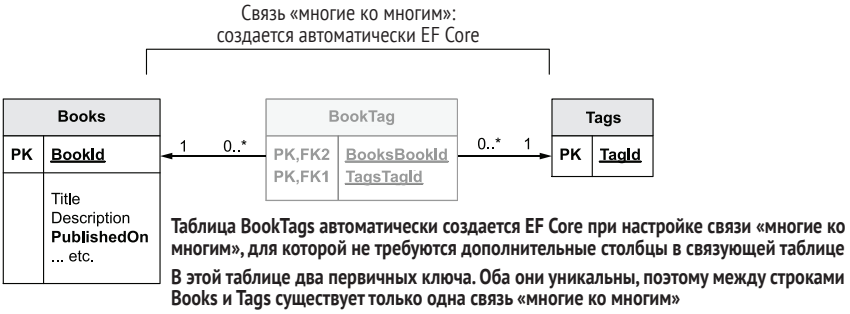


Рис. 2.4 Вы создаете таблицы Books и Tags, и EF Core распознает связь «многие ко многим» между таблицами Books и Tags. EF Core автоматически создает связующую таблицу, необходимую для настройки связи «многие ко многим»

ПРИМЕЧАНИЕ В главе 8 рассматриваются различные способы установления связи «многие ко многим».

Вот как это будет выглядеть на экране:

Категории: Microsoft .NET, Web

2.1.2 Другие типы связей, не описанные в этой главе

Три типа связей, которые я рассмотрел в разделе 2.1.1, являются основными связями, которые вы будете использовать: один к одному,

один ко многим и многие ко многим. Но в EF Core есть и другие варианты. Вот краткое изложение того, что будет дальше в главе 8:

- *собственный тип* (*Owned Type class*) – полезен для добавления сгруппированных данных, таких как класс `Address`, в класс сущности. Класс `Address` связан с основной сущностью, но ваш код может скопировать класс `Address`, вместо того чтобы копировать улицу (`Street`), город (`City`), штат (`State`) и связанные свойства по отдельности;
- *разделение таблицы* (*Table splitting*) – отображает несколько классов в одну таблицу. У вас может быть сводный класс с основными свойствами в нем и подробный класс, содержащий, например, все данные, что позволит быстрее загружать сводные данные;
- *таблица на иерархию* (*TPH*) – полезно для схожих групп данных. Если у вас много данных с небольшими отличиями, например список животных, у вас может быть базовый класс `Animal`, от которого могут наследовать классы `Dog`, `Cat` и `Snake`, со свойствами по каждому типу, например `LengthOfTail` для `Dog` и `Cat` и флаг `Venomous` для `Snake`. EF Core отображает все классы в одну таблицу, что может быть более эффективно;
- *таблица на тип* (*TPT*) – полезно для групп данных с разными данными. Данный подход, представленный в EF Core 5, является противоположностью TPH – у каждого класса своя таблица. Следуя примеру с классом `Animal`, при использовании TPT классы `Dog`, `Cat` и `Snake` будут отображаться в три разные таблицы.

Эти четыре шаблона связей встроены в EF Core, чтобы можно было оптимизировать способ обработки или хранения данных в базе данных. Но есть еще один тип связей, которому не нужны специальные команды EF Core для реализации: *иерархические* данные. Типичный пример иерархических данных – это класс сотрудника (`Employee`), имеющий связь, указывающую на руководителя, который в свою очередь также является сотрудником. EF Core использует те же подходы, что и «один к одному» и «один ко многим» для обеспечения иерархических связей; подробнее об этой связи мы поговорим в главах 6 и 8.

2.1.3 База данных – все таблицы

На рис. 2.5 показана база данных приложения Book App, которую вы будете использовать для примеров из этой главы и главы 3. В ней содержатся все таблицы, которые я описывал до сих пор, включая все столбцы и связи из таблицы Books.

> **ПРИМЕЧАНИЕ** В этой схеме базы данных используется та же структура и термины, что и в главе 1: ПК означает *первичный ключ*, а ВК – *внешний ключ*.

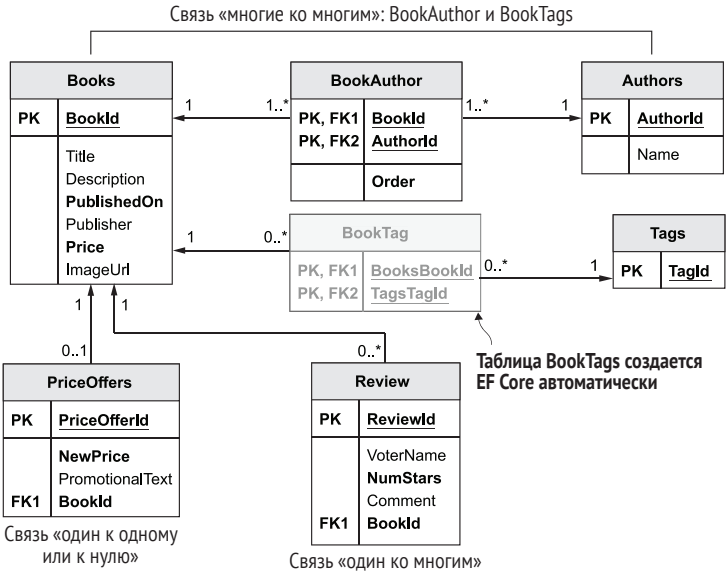


Рис. 2.5 Полная схема реляционной базы данных для приложения Book App, показывающая все таблицы и их столбцы, используемые для хранения информации о книгах. Вы создаете классы для отображения во все таблицы, которые вы видите на этом рисунке, кроме таблицы BookTags (выделенной бледно-серым). EF Core автоматически создает таблицу BookTags, когда обнаруживает прямую связь «многие ко многим» между таблицами Books и Tags

Рис. 2.6 Описание отдельной книги, показывающее, какая таблица базы данных предоставляет каждый фрагмент информации. Как видно, для создания этого представления требуется информация из всех пяти таблиц базы данных. В этой главе вы создадите код для вывода такой информации на экран с сортировкой, фильтрацией и разбиением по страницам для создания правильного приложения электронной коммерции

Чтобы помочь вам разобраться в этой базе данных, на рис. 2.6 показан вывод списка книг на экран, но здесь только одна книга. Как видите, приложению Book App требуется доступ к каждой таблице из базы данных для создания списка книг (рис. 2.10 в разделе 2.6). Позже я покажу вам тот же самый вариант, но с запросом, который предоставляет каждый элемент.

Скачивание и запуск приложения из репозитория Git

Если вы хотите загрузить код приложения Book App и запустить его локально, выполните действия, указанные на одноименной боковой панели из раздела 1.6.2. Ветка master содержит весь код для первой части книги, которая включает проект ASP.NET Core BookApp.

2.1.4 Классы, которые EF Core отображает в базу данных

Я создал пять классов .NET для отображения в шесть таблиц базы данных. Это классы `Book`, `PriceOffer`, `Review`, `Tag`, `Author` и `BookAuthor` для связующей таблицы «многие ко многим». Они называются *классами сущностей*, чтобы показать, что они отображаются EF Core в базу данных. С точки зрения программного обеспечения в них нет ничего особенного. Это обычные классы .NET, которые иногда называют *Plain Old CLR Object* (POCO). Термин *класс сущности* определяет класс, который EF Core отображает в базу данных.

Основным классом сущности является класс `Book`, показанный в следующем листинге. Видно, что он ссылается на один экземпляр класса `PriceOffer` и коллекции экземпляров классов `Review`, `Tag` и `BookAuthor`. Последняя связывает данные книги с одним или несколькими классами `Author`, содержащими фамилию и имя автора.

Листинг 2.1 Класс Book, отображаемый в таблицу Books в базе данных

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime PublishedOn { get; set; }
    public string Publisher { get; set; }
    public decimal Price { get; set; }
    public string ImageUrl { get; set; }

    //-----------------------------------------------
    //Связи;

    public PriceOffer Promotion { get; set; }
    public ICollection<Review> Reviews { get; set; }
    public ICollection<Tag> Tags { get; set; }
    public ICollection<BookAuthor>
      AuthorsLink { get; set; }
}
```

Класс Book содержит основную информацию о книге

Мы используем конфигурацию EF Core «По соглашению» для определения первичного ключа этого класса сущности, поэтому применяем <ClassName>Id, а поскольку свойство имеет тип int, EF Core предполагает, что база данных будет использовать SQL-команду IDENTITY для создания уникального ключа при добавлении новой строки

Ссылка на необязательную связь «один к одному» PriceOffer

Количество отзывов на книгу может варьироваться от нуля до множества

Автоматическая связь «многие ко многим» в EF Core 5 с классом сущности Tag

Предоставляет ссылку на связующую таблицу «многие ко многим», которая связана с таблицей Authors этой книги

ПРИМЕЧАНИЕ В первой части классы сущностей используют конструктор по умолчанию (без параметров). Если вы хотите создать конкретные конструкторы для какого-либо класса сущности, следует иметь в виду, что EF Core может использовать ваш конструктор при чтении и создании экземпляра класса сущности. Об этом рассказывается в разделе 6.1.11.

Для простоты при моделировании базы данных используется подход «По соглашению» (By Convention). Мы используем его при именовании свойств, которые содержат первичный ключ и внешние ключи в каждом из классов сущностей. Кроме того, тип навигационных свойств, например `ICollection<Review> Reviews`, определяет, какого рода связь нам нужна. Поскольку свойство `Reviews` относится к типу `ICollection.<Review>`, это связь «один ко многим». В главах 7 и 8 описаны другие подходы к настройке модели базы данных EF Core.

ДОПОЛНИТЕЛЬНОЕ ПРИМЕЧАНИЕ В приложении Book App, когда у класса есть навигационные свойства, которые являются коллекциями, я использую тип `ICollection<T>`, потому что новая возможность немедленной загрузки с сортировкой (см. раздел 2.4.1) может возвращать отсортированную коллекцию, а определение по умолчанию `HashSet` не поддерживает сортировку. Но если навигационное свойство содержит большое количество элементов, отказ от `HashSet` приводит к снижению производительности. Об этом рассказывается в главе 14.

Что произойдет, если вы захотите получить доступ к существующей базе данных?

Примеры из этой книги показывают, как определить и создать базу данных с помощью EF Core, потому что самая сложная ситуация – когда нужно разбираться во всех конфигурационных параметрах. Но получить доступ к существующей базе данных намного проще, потому что EF Core может создать класс DbContext приложения и все классы сущностей за вас, используя *обратное проектирование* (*Reverse engineering*), описанное в разделе 9.7.

Другая возможность заключается в том, что вы не хотите, чтобы EF Core менял структуру базы данных, а хотите решить эту задачу самостоятельно, например с помощью сценария изменения SQL или инструмента развертывания базы данных. Я рассмотрю этот подход в разделе 9.6.2.

2.2 Создание DbContext

Чтобы получить доступ к базе данных, необходимо сделать следующее:

1. Определить DbContext своего приложения, создав класс, унаследованный от класса EF Core `DbContext`.
2. Создавать экземпляр этого класса каждый раз, когда вы хотите получить доступ к базе данных.

Все запросы к базе данных, которые вы увидите далее в этой главе, используют эти шаги. Они будут подробно описаны в следующих разделах.

2.2.1 Определение DbContext приложения: EfCoreContext

Ключевым классом, необходимым для использования EF Core, является DbContext. Он определяется путем наследования от класса EF Core `DbContext` и добавления различных свойств, позволяющих программе получать доступ к таблицам базы данных. Он также содержит методы, которые можно переопределить, чтобы получить доступ к другим функциональным возможностям EF Core, например к настройке моделирования базы данных. На рис. 2.7 представлен обзор DbContext приложения Book App с указанием всех важных частей.

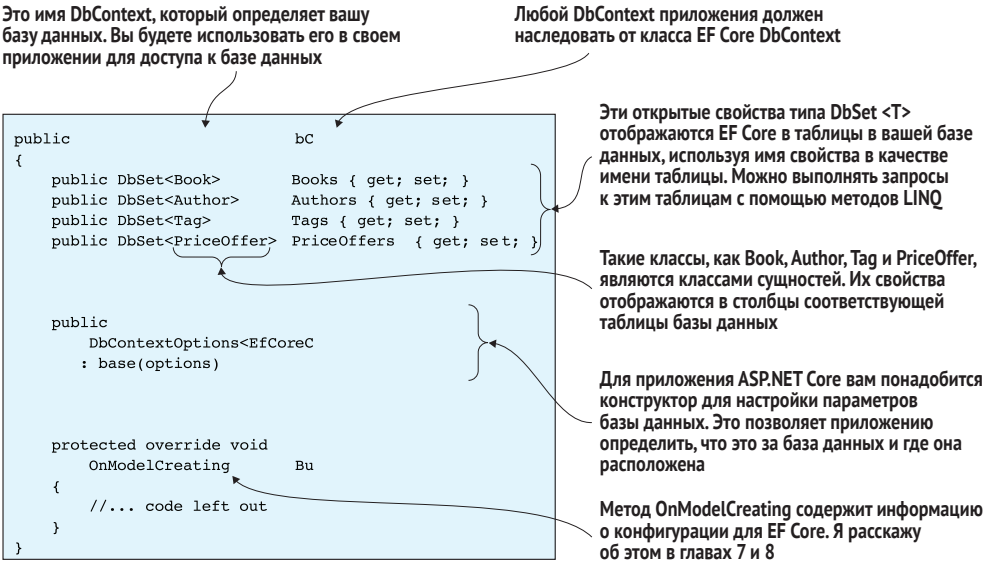


Рис. 2.7 DbContext приложения – это ключевой класс при доступе к базе данных. На этом рисунке показаны основные части DbContext, начиная с наследования от класса EF Core DbContext, который предоставляет большое количество кода и функциональных возможностей. Нужно добавить свойства с типом `DbSet<T>`, которые отображают ваши классы в таблицу базы данных с тем же именем, что и имя свойства, которое вы используете. Остальные части – это конструктор, который занимается настройкой параметров базы данных, и метод `OnModelCreating`, который можно переопределить, чтобы добавить собственные команды конфигурации и настроить базу данных так, как вы хотите

Здесь следует отметить, что DbContext в приложении Book App не включает свойства DbSet<T> для классов сущностей Review и BookAuthor. В этом приложении доступ к обоим классам сущностей осуществляется не напрямую, а через навигационные свойства класса Book, как вы увидите в разделе 2.4.

ПРИМЕЧАНИЕ Я пропускаю настройку моделирования базы данных, которая выполняется в методе OnModelCreating DbContext приложения. В главах 7 и 8 подробно описано, как смоделировать базу данных.

2.2.2 Создание экземпляра DbContext приложения

В главе 1 показано, как настроить DbContext приложения, переопределив его метод OnConfiguring. Но у этого подхода есть и обратная сторона: фиксированная строка подключения. В этой главе мы воспользуемся другим подходом, поскольку нам нужно использовать различные базы данных для разработки и модульного тестирования. Мы воспользуемся методом, предоставляющим эту базу данных через конструктор DbContext.

В листинге 2.2 представлены параметры для базы данных во время создания DbContext, называемого EfCoreContext. Честно говоря, этот листинг основан на том, что я использую в главе, посвященной модульному тестированию (глава 17), потому что это позволяет показать каждый этап создания экземпляра DbContext приложения. В главе 5, посвященной использованию EF Core в приложении ASP.NET Core, представлен более мощный способ создания DbContext с помощью внедрения зависимостей.

Листинг 2.2 Создание экземпляра DbContext приложения для доступа к базе данных

```
const string connection =
     "Data Source=(localdb)\\mssqllocaldb;"+
     "Database=EfCoreInActionDb.Chapter02;"+
     "Integrated Security=True;";
var optionsBuilder =
    new DbContextOptionsBuilder
           <EfCoreContext>();

optionsBuilder.UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{
    var bookCount = context.Books.Count();
    //... и т.д.
}
```

Строка подключения, формат которой зависит от типа провайдера базы данных и используемого хостинга

Вам понадобится экземпляр EF Core DbContextOptionsBuilder<>, чтобы установить нужные параметры

Вы получаете доступ к базе данных SQL Server и используете метод UseSqlServer из библиотеки Microsoft.EntityFrameworkCore.SqlServer. Этому методу требуется строка подключения к базе данных

Создает важнейший EfCoreContext, используя настроенные вами параметры. Используется оператор using, потому что этот DbContext освобождаемый

Использует DbContext, чтобы узнать количество книг в базе данных

В конце этого листинга в операторе `using` создается экземпляр `EfCoreContext`, и поскольку он реализует интерфейс `IDisposable`, это означает, что после выхода из данного блока этот экземпляр будет уничтожен. Итак, с этого момента, если вы видите переменную `context`, она была создана с использованием кода из листинга 2.2 или аналогичного подхода.

2.2.3 Создание базы данных для своего приложения

Есть несколько способов создать базу данных с помощью EF Core, но обычно для этого используется миграция. Она применяет DbContext приложения и классы сущностей, подобные тем, что я описал, в качестве модели для структуры базы данных. Сначала команда `Add-Migration` моделирует вашу базу данных, а затем, используя эту модель, строит команды для создания базы данных, соответствующей этой модели.

> **СОВЕТ** Если вы клонировали репозиторий Git из этой книги (http://mng.bz/XdlG), то можно увидеть, как выглядит миграция, просмотрев папку Migration из проекта DataLayer. Кроме того, в проекты DataLayer и BookApp были добавлены все нужные пакеты NuGet, чтобы можно было создавать миграции и применять их к базе данных SQL Server.

Помимо создания базы данных, у миграций есть одно замечательное свойство, которое заключается в том, что они могут обновлять базу данных, когда вы вносите изменения в программное обеспечение. Если изменить классы сущностей или любую конфигурацию DbContext приложения, команда `Add-Migration` создаст набор команд для обновления существующей базы данных.

Вот шаги, которые необходимо выполнить, чтобы добавить миграцию и создать или перенести базу данных. Этот процесс основан на приложении ASP.NET Core (см. главу 5 для получения дополнительной информации об ASP.NET Core) с DbContext в отдельном проекте и разработан с помощью Visual Studio. (О других вариантах будет рассказано в главе 9.)

1. Проекту, содержащему ваш DbContext, требуется пакет NuGet Microsoft.EntityFrameworkCore.SqlServer или другой провайдер базы данных, если вы используете иную базу данных.
2. Проекту ASP.NET Core нужны следующие пакеты NuGet:
   a. Microsoft.EntityFrameworkCore.SqlServer (или тот же провайдер базы данных, что и на этапе 1);
   b. Microsoft.EntityFrameworkCore.Tools.
3. Класс ASP.NET Core `Startup` содержит команды для добавления провайдера базы данных EF Core, а файл appsettings.json содержит строку подключения для базы данных, которую вы хотите

создать или обновить. (EF Core использует ASP.NET Core методы `CreateHostBuilder(args).Build()` для получения действительного экземпляра DbContext.)

4. В Visual Studio откройте консоль диспетчера пакетов (PMC), выбрав **Tools** > **NuGet Package Manager** > **Package Manager Console** (Инструменты > Диспетчер пакетов NuGet > Консоль диспетчера пакетов).
5. В окне PMC убедитесь, что проект по умолчанию – это ваш проект ASP.NET Core.
6. В PMC запустите команду `Add-Migration MyMigrationName -Project Data-Layer`. Эта команда создает набор классов, которые переносят базу данных из ее текущего состояния в состояние, соответствующее DbContext приложения и классам сущностей на момент выполнения команды. (`MyMigrationName` – это имя, которое будет использоваться для миграции.)
7. Выполните команду `Update-Database`, чтобы применить миграцию, созданную с помощью команды `Add- Migration`, к своей базе данных. Если базы данных нет, то `Update-Database` создаст ее. Если база данных существует, то команда проверит, применена ли к ней эта миграция, и, если какие-либо миграции отсутствуют, применит их.

ПРИМЕЧАНИЕ Для выполнения этих команд (см. http://mng.bz/454w) также можно использовать интерфейс командной строки .NET Core. В главе 9 перечислены версии команд миграции для Visual Studio и интерфейса командной строки.

Альтернатива использованию команды `Update-Database` – вызов метода `context.Database.Migrate` в коде запуска вашего приложения. Этот подход особенно полезен для размещенного веб-приложения ASP.NET Core; глава 5 посвящена описанию этого варианта, включая некоторые его ограничения.

ПРИМЕЧАНИЕ В главе 9 подробно рассматривается миграция, а также другие способы изменить структуру базы данных (также именуемую как *схема* базы данных).

2.3 Разбираемся с запросами к базе данных

Теперь можно приступить к изучению того, как выполнять запрос к базе данных с помощью EF Core. На рис. 2.8 показан пример запроса к базе данных EF Core, где выделены три основные части запроса.

ЧТОБЫ СЭКОНОМИТЬ ВРЕМЯ Если вы знакомы с EF и/или LINQ, то можно пропустить этот раздел.

Рис. 2.8 Три части запроса к базе данных EF Core с примером кода. Вы познакомитесь с этим типом LINQ-выражений, которые являются основным строительным блоком всех запросов

Команда, показанная на рис. 2.8, состоит из нескольких методов, следующих один за другим. Эта структура известна как *текучий интерфейс* (*fluent interface*). Текучие интерфейсы, подобные этому, работают логически и интуитивно понятно, что облегчает их чтение. Три части этой команды описаны в следующих разделах.

ПРИМЕЧАНИЕ Команда LINQ, изображенная на рис. 2.8, известна как метод LINQ, или синтаксис лямбда-выражений. Для написания команд LINQ с EF Core можно использовать и другой формат: синтаксис запросов. Их описание приводится в приложении А.

2.3.1 Доступ к свойству DbContext приложения

Первая часть команды подключается к базе данных через EF Core. Самый распространенный способ обращения к таблице базы данных – использование свойства `DbSet<T>` в DbContext приложения, как показано на рис. 2.7.

Вы будете использовать этот доступ к свойству DbContext на протяжении всей главы, но в последующих главах описаны другие способы получения класса или свойства. Основная идея та же: нужно начинать с чего-то, что подключено к базе данных через EF Core.

2.3.2 Серия команд LINQ / EF Core

Основная часть команды – это набор методов LINQ и/или EF Core, которые создают нужный вам тип запроса. Запрос LINQ может варьироваться от сверхпростого до довольно сложного. Эта глава начинается с простых примеров, но в конце вы узнаете, как создавать сложные запросы.

ПРИМЕЧАНИЕ *Вам будет необходимо изучить LINQ, поскольку EF Core использует команды LINQ для доступа к базе данных.* В приложении приводится краткий обзор LINQ, а также на эту тему доступно много онлайн-ресурсов; см. http://mng.bz/j4Qx.

2.3.3 Команда выполнения

Последняя часть команды рассказывает кое-что о LINQ. Пока в конце последовательности команд LINQ не будет применена последняя

команда выполнения, LINQ хранит команды в т. н. *дереве выражений* (см. раздел A.2.2). Это означает, что обращения к данным еще не происходит. EF Core может преобразовывать дерево выражений в правильные команды для используемой вами базы данных. В EF Core запрос выполняется к базе данных, когда

- он перечисляется оператором `foreach`;
- он перечисляется операцией преобразования в коллекцию, такой как `ToArray`, `ToDictionary`, `ToList`, `ToListAsync` и т. д.;
- такие операторы LINQ, как `First` или `Any`, указываются во внешней части запроса.

Вы будете использовать определенные команды EF Core, такие как `Load`, при явной загрузке связи далее в этой главе.

На данном этапе ваш LINQ-запрос будет преобразован в команды базы данных и отправлен в базу данных. Если вы хотите создавать высокопроизводительные запросы к базе данных, нужно, чтобы все ваши LINQ-команды для фильтрации, сортировки, разбиения по страницам и т. д. поступали до того, как вы вызовите команду выполнения. Таким образом, все эти команды будут выполняться внутри базы данных, что повышает производительность запроса. Вы увидите этот подход в действии в разделе 2.8, когда мы создадим запрос для фильтрации, сортировки и разбиения списка книг на страницы в базе данных для отображения пользователю.

2.3.4 Два типа запросов к базе данных

Запрос к базе данных на рис. 2.8 – это то, что я называю *обычным* запросом, также известным как запрос *с доступом на чтение и запись*. Он считывает данные из базы данных таким образом, что их можно обновить (см. главу 3) или использовать в качестве существующей связи для новой записи, например создания новой книги с существующим автором (см. раздел 6.2.2).

Другой тип запроса – это запрос AsNoTracking, также известный как запрос с доступом только на чтение. Он содержит метод EF Core `AsNoTracking`, добавленный в запрос LINQ (см. следующий фрагмент кода). Помимо того что он превращает запрос в запрос с доступом только на чтение, метод `AsNoTracking` повышает производительность запроса за счет отключения определенных функциональных возможностей EF Core; см. раздел 6.12 для получения дополнительной информации:

```
context.Books.AsNoTracking()
      .Where(p => p.Title.StartsWith("Quantum")).ToList();
```

ПРИМЕЧАНИЕ В разделе 6.1.2 приведен подробный список различий между обычным запросом на чтение и запись и запросом AsNoTracking с доступом только на чтение.

2.4 Загрузка связанных данных

Я показал вам класс сущности Book, у которого есть связи с тремя другими классами: PriceOffer, Review и BookAuthor. Теперь я хочу объяснить, как вы можете получить доступ к этим связанным данным. Можно загружать данные, используя четыре способа: немедленную, явную, выборочную и отложенную. Однако, прежде чем я расскажу об этих подходах, вы должны знать, что EF Core не будет загружать никакие связи в классе сущности, если вы об этом не попросите. Если загрузить класс Book, то все свойства связи в классе сущности Book (Promotion, Reviews и AuthorsLink) по умолчанию будут иметь значение null.

Такое поведение по умолчанию, когда связи не загружаются, является правильным, потому что это означает, что EF Core минимизирует доступ к базе данных. Если вы хотите загрузить связь, то нужно добавить код, сообщающий EF Core об этом. В следующих разделах описываются четыре подхода, которые заставляют EF Core загружать связанные данные.

2.4.1 Немедленная загрузка: загрузка связей с первичным классом сущности

Первый подход к загрузке связанных данных – это *немедленная загрузка*, которая велит EF Core загружать связи в том же запросе, который загружает основной класс сущности.

Немедленная загрузка определяется двумя методами: Include и ThenInclude. В следующем листинге показаны загрузка первой строки таблицы Book как экземпляра класса сущности Book и немедленная загрузка единственной связи, Reviews.

Листинг 2.3 Немедленная загрузка первой книги с соответствующей связью Reviews

```
var firstBook = context.Books
    .Include(book => book.Reviews)
    .FirstOrDefault();
```

Получает коллекцию экземпляров класса Review, которая может быть пустой

Выбирает первую книгу или возвращает null, если в базе нет книг

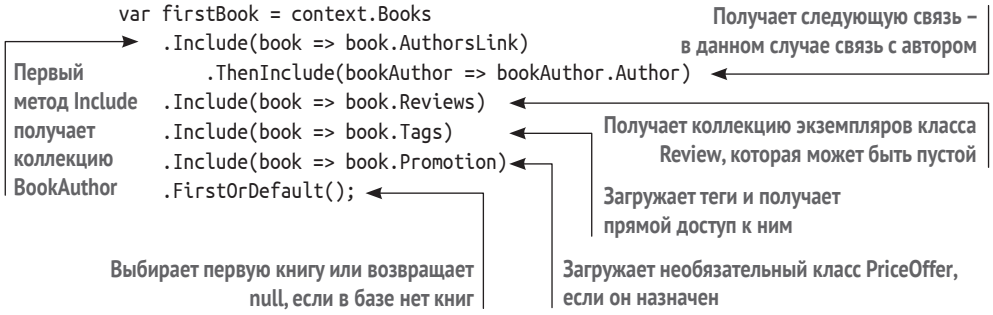
Если посмотреть на SQL-команду, которую создает этот запрос EF Core, как показано в следующем фрагменте кода, то можно увидеть две команды. Первая команда загружает первую строку в таблицу Books. Вторая загружает отзывы, где внешний ключ BookId имеет то же значение, что и первичный ключ первой строки Books:

```
SELECT "t"."BookId", "t"."Description", "t"."ImageUrl",
    "t"."Price", "t"."PublishedOn", "t"."Publisher",
       "t"."Title", "r"."ReviewId", "r"."BookId",
            "r"."Comment", "r"."NumStars", "r"."VoterName"
```

```
FROM (
    SELECT "b"."BookId", "b"."Description", "b"."ImageUrl",
        "b"."Price", "b"."PublishedOn", "b"."Publisher", "b"."Title"
    FROM "Books" AS "b"
    LIMIT 1
) AS "t"
LEFT JOIN "Review" AS "r" ON "t"."BookId" = "r"."BookId"
ORDER BY "t"."BookId", "r"."ReviewId"
```

Теперь рассмотрим более сложный пример. В следующем листинге показан запрос на получение первой книги с немедленной загрузкой всех ее связей – в данном случае `AuthorsLink` и таблицы `Author` второго уровня, `Reviews` и необязательного класса `Promotion`.

Листинг 2.4 Немедленная загрузка класса Book и всех связанных данных



В листинге показано использование метода немедленной загрузки `Include` для получения связи `AuthorsLink`. Это связь первого уровня, на которую ссылаются непосредственно из загруженного вами класса сущности. После этого метода следует метод `ThenInclude` для загрузки связи второго уровня – в данном случае таблицы `Author` на другом конце связи через таблицу `BookAuthor`. Этот шаблон, включающий метод `Include`, за которым следует `ThenInclude`, – стандартный способ доступа к связям, которые идут глубже, чем связь первого уровня. С помощью нескольких методов `ThenInclude`, следующих один за другим, можно зайти настолько глубоко, насколько требуется.

Если вы используете прямое связывание, представленное в EF Core 5, для загрузки связи второго уровня метод `ThenInclude` не требуется, потому что свойство напрямую обращается к другому концу связи «многие ко многим» через свойство `Tags` типа `ICollection<Tag>`. Такой подход может упростить использование связи «многие ко многим», если вам не нужны данные из связующей таблицы, например свойство `Order` из класса сущности `BookAuthor`, используемое для надлежащего упорядочивания фамилий и имен авторов книги.

> **EF6** Немедленная загрузка в EF Core аналогична загрузке в EF6.x, но в EF6.x нет метода `ThenInclude`. В результате код `Include/ThenInclude`, использованный в листинге 2.4, в EF6.x будет выглядеть так: `context.Books.Include(book => book.AuthorLink.Select(bookAuthor => bookAuthor.Author)`.

Если связанных данных не существует (например, необязательный класс `PriceOffer`, на который указывает свойство `Promotion` в классе `Book`), то `Include` не завершится ошибкой; он просто ничего не загрузит или, в случае с коллекциями, вернет пустую коллекцию (обычную коллекцию без элементов). То же правило применимо и к `ThenInclude`: если предыдущий метод `Include` или `ThenInclude` были пустыми, то последующие методы `ThenInclude` игнорируются. Если вы не используете метод `Include` для коллекции, то по умолчанию значение будет равно null.

Преимущество немедленной загрузки заключается в том, что EF Core будет загружать все данные, на которые ссылаются методы `Include` и `ThenInclude`, эффективным способом, используя минимум обращений к базе данных. Я считаю, что данный тип загрузки полезен в реляционных обновлениях, когда нужно обновить существующие связи; этой теме посвящена глава 3. Я также считаю, что немедленная загрузка полезна в бизнес-логике; эта тема более подробно рассматривается в главе 4.

Недостаток данного подхода состоит в том, что при немедленной загрузке загружаются *все* данные, даже если они вам не нужны. Например, для отображения списка книг не требуется описание книги, которое может быть довольно большим.

СОРТИРОВКА И ФИЛЬТРАЦИЯ ПРИ ИСПОЛЬЗОВАНИИ МЕТОДА INCLUDE И/ИЛИ THENINCLUDE

В EF Core 5 добавлена возможность сортировки или фильтрации связанных сущностей при использовании методов `Include` или `ThenInclude`. Это полезно, если вы хотите загрузить только подмножество связанных данных (например, только отзывы с пятью звездами) и/или упорядочить включенные сущности (например, упорядочить коллекцию `AuthorsLink` по свойству `Order`). Единственные команды LINQ, которые можно использовать в методах `Include` или `ThenInclude`, – это `Where`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `Skip` и `Take`, но этого достаточно для сортировки и фильтрации.

В следующем листинге показан тот же код, что и в листинге 2.4, но коллекция `AuthorsLink` сортируется по свойству `Order`, а коллекция `Reviews` фильтруется для загрузки только тех отзывов, где `NumStars` равно 5.

Листинг 2.5 Сортировка и фильтрация при использовании метода Include или ThenInclude

```
var firstBook = context.Books
    .Include(book => book.AuthorsLink
        .OrderBy(bookAuthor => bookAuthor.Order))
        .ThenInclude(bookAuthor => bookAuthor.Author)
    .Include(book => book.Reviews
        .Where(review => review.NumStars == 5))
    .Include(book => book.Promotion)
    .First();
```

Пример сортировки: при немедленной загрузке коллекции AuthorsLink вы сортируете BookAuthors так, чтобы авторы располагались в правильном порядке для отображения

Пример фильтра. Здесь загружаются только отзывы с рейтингом 5 звезд

2.1.2 Явная загрузка: загрузка связей после первичного класса сущности

Второй подход к загрузке данных – это *явная загрузка*. После загрузки первичного класса сущности можно явно загрузить любые другие связи, которые захотите. Листинг 2.6 выполняет ту же работу, что и листинг 2.4, используя явную загрузку. Сначала загружается Book; после этого используются команды явной загрузки для чтения всех связей.

Листинг 2.6 Явная загрузка класса Book и связанных данных

```
var firstBook = context.Books.First();
context.Entry(firstBook)
    .Collection(book => book.AuthorsLink).Load();
foreach (var authorLink in firstBook.AuthorsLink)
{
    context.Entry(authorLink)
        .Reference(bookAuthor =>
            bookAuthor.Author).Load();
}
context.Entry(firstBook)
    .Collection(book => book.Tags).Load();
context.Entry(firstBook)
    .Reference(book => book.Promotion).Load();
```

Читает первую книгу

Явно загружает связующую таблицу, BookAuthor

Чтобы загрузить всех возможных авторов, код должен перебрать все записи BookAuthor ...

... и загрузить все связанные классы Author

Загружает все отзывы

Загружает Tags

Загружает необязательный класс PriceOffer

В качестве альтернативы можно использовать явную загрузку для применения запроса к связи вместо ее загрузки. В листинге 2.7 показано использование метода явной загрузки Query для получения количества отзывов и загрузки рейтинга каждого отзыва. После метода Query можно использовать любую стандартную команду LINQ, например Where или OrderBy.

Листинг 2.7 Явная загрузка класса Book с уточненным набором связанных данных

```
var firstBook = context.Books.First();          ← Читает данные о первой книге
var numReviews = context.Entry(firstBook)
    .Collection(book => book.Reviews)           | Выполняет запрос для подсчета
    .Query().Count();                           | отзывов по этой книге
var starRatings = context.Entry(firstBook)
    .Collection(book => book.Reviews)           | Выполняет запрос на получение
    .Query().Select(review => review.NumStars)  | всех рейтингов книги
    .ToList();
```

Преимущество явной загрузки заключается в том, что можно загрузить связанные данные класса сущности позже. Я обнаружил, что этот метод полезен при использовании библиотеки, которая загружает только основной класс сущности, а мне нужна одна из его связей. Явная загрузка также может быть полезна, когда вам нужны связанные данные только при некоторых обстоятельствах. Она также может оказаться полезной в сложной бизнес-логике, потому что можно оставить загрузку конкретных связей тем частям бизнес-логики, которые в этом нуждаются.

Недостатком явной загрузки является большее количество обращений к базе данных, что может быть неэффективным. Если вы заранее знаете, какие данные вам нужны, то немедленная загрузка обычно более эффективна, потому что для загрузки связей требуется меньше обращений к базе данных.

2.4.3 Выборочная загрузка: загрузка определенных частей первичного класса сущности и любых связей

Третий подход к загрузке данных заключается в использовании метода LINQ `Select` для выбора нужных данных, который я называю *выборочной загрузкой*. В следующем листинге показано использование метода `Select`, чтобы выбрать несколько стандартных свойств из класса `Book` и выполнить определенный код внутри запроса, чтобы получить количество отзывов покупателей на эту книгу.

Листинг 2.8 Использование метода Select с классом Book для выбора определенных свойств и вычисления

```
var books = context.Books
    .Select(book => new                  ← Использует ключевое слово LINQ Select и создает
        {                                  анонимный тип для хранения результатов
Простые копии | book.Title,
пары свойств  | book.Price,
              NumReviews                 | Выполняет запрос для подсчета
                  = book.Reviews.Count,  | количества отзывов
        }
    ).ToList();
```

Преимущество этого подхода состоит в том, что здесь загружаются только нужные вам данные. Если вам не нужны все данные, то, возможно, это более эффективный вариант. В листинге 2.8 требуется только одна команда SQL, `SELECT`, для получения всех этих данных, что также эффективно с точки зрения обращений к базе данных. EF Core превращает часть запроса `p.Reviews.Count` в команду SQL, поэтому подсчет выполняется внутри базы данных, что видно из следующего фрагмента SQL, созданного EF Core:

```
SELECT "b"."Title", "b"."Price", (
    SELECT COUNT(*)
    FROM "Review" AS "r"
    WHERE "b"."BookId" = "r"."BookId") AS "NumReviews"
FROM "Books" AS "b"
```

Недостаток этого подхода состоит в том, что нужно писать код для каждого свойства или подсчета. В разделе 7.15.4 приводится способ автоматизации этого процесса.

ПРИМЕЧАНИЕ В разделе 2.6 есть гораздо более сложный пример выборочной загрузки, который мы будем использовать для создания высокопроизводительного запроса на получение списка книг для приложения Book App.

2.4.4 Отложенная загрузка: загрузка связанных данных по мере необходимости

Отложенная загрузка упрощает написание запросов, но плохо влияет на производительность базы данных. Она требует некоторых изменений в DbContext или классах сущностей, но после внесения этих изменений чтение данных становится легким; если вы обращаетесь к незагруженному навигационному свойству, то EF Core выполнит запрос к базе данных для загрузки этого свойства.

Отложенную загрузку можно настроить двумя способами:

- добавить библиотеку Microsoft.EntityFrameworkCore.Proxies при настройке DbContext;
- внедрить метод отложенной загрузки в класс сущности, используя конструктор.

Первый вариант простой, но не дает настроить отложенную загрузку для отдельных связей. Второй вариант требует написания большего количества кода, но позволяет выбрать, какие связи используют отложенную загрузку. В этой главе я объясню только первый вариант, потому что он простой, а второй вариант оставлю для главы 6 (раздел 6.1.10), потому что в нем используются концепции, которые мы еще не рассматривали, например внедрение зависимостей.

ПРИМЕЧАНИЕ Если хотите увидеть все варианты отложенной загрузки прямо сейчас, посетите страницу https://docs.microsoft.com/en-us/ef/core/querying/related-data/lazy.

Чтобы настроить простой подход с отложенной загрузкой, нужно сделать две вещи:

- добавить ключевое слово `virtual` перед *каждым* свойством, которое является связью;
- добавить метод `UseLazyLoadingProxies` при настройке DbContext.

Таким образом, преобразованный тип сущности Book для простого подхода с отложенной загрузкой будет выглядеть, как показано в следующем фрагменте кода с ключевым словом `virtual`, добавленным к навигационным свойствам:

```
public class BookLazy
{
    public int BookLazyId { get; set; }
    //… Другие свойства не указаны для ясности;

    public virtual PriceOffer Promotion { get; set; }
    public virtual ICollection<Review> Reviews { get; set; }
    public virtual ICollection<BookAuthor> AuthorsLink { get; set; }
}
```

Использование библиотеки EF Core Proxy имеет ограничение: нужно сделать все реляционные свойства виртуальными; в противном случае EF Core выбросит исключение при использовании DbContext.

Вторая часть – это добавление библиотеки Proxy в приложение, которое настраивает DbContext. После этого добавляем `UseLazyLoadingProxies` в настройку DbContext. В следующем фрагменте кода показано добавление метода `UseLazyLoadingProxies` к коду настройки DbContext, который мы видели в листинге 2.2:

```
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();
optionsBuilder
    .UseLazyLoadingProxies()
    .UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
```

После настройки отложенной загрузки в классах сущностей и в способе создания DbContext загружать связи становится просто; вам не нужны дополнительные методы `Include` в запросе, потому что данные загружаются из базы данных, когда код обращается к этому свойству связи. В листинге 2.9 показана отложенная загрузка свойства `Reviews`.

Листинг 2.9 Отложенная загрузка навигационного свойства Reviews

```
var book = context.BookLazy.Single();
var reviews = book.Reviews.ToList();
```

Получает экземпляр класса сущности BookLazy, свойство Reviews которого настроено на использование отложенной загрузки

При доступе к свойству Reviews EF Core прочитает отзывы из базы данных

В листинге 2.9 создается два запроса к базе данных. Первый запрос загружает данные `BookLazy` без каких-либо свойств, а второй – при доступе к свойству `Reviews` класса `BookLazy`.

Многие разработчики считают отложенную загрузку полезной, но я избегаю ее из-за проблем с производительностью. Каждый доступ к серверу базы данных связан с дополнительными затратами по времени, поэтому лучший подход – минимизировать количество обращений к серверу. Но отложенная загрузка (и явная загрузка) могут создавать множество обращений к базе данных, замедляя выполнение запроса и заставляя сервер базы данных работать интенсивнее. Смотрите раздел 14.5.1, где приводится сравнение четырех типов загрузки связанных данных.

СОВЕТ Даже если вы настроили реляционное свойство для отложенной загрузки, можно добиться повышения производительности за счет добавления метода `Include` к виртуальному реляционному свойству. Отложенная загрузка увидит, что свойство было загружено, и не станет загружать его снова. Например, если изменить первую строку из листинга 2.9 на `context.BookLazy.Include(book => book.Reviews).Single()`, то это уменьшит количество запросов к базе данных до одного.

2.5 Использование вычисления на стороне клиента: адаптация данных на последнем этапе запроса

Все запросы, которые вы видели до сих пор, EF Core может преобразовать в команды, которые можно выполнять на сервере базы данных. Но в EF Core существует *вычисление на стороне клиента*, позволяющее выполнять код на последнем этапе запроса (последняя часть с `Select` в запросе), который нельзя преобразовать в команды базы данных. EF Core выполняет эти команды, не поддерживаемые сервером, после того как данные вернутся из базы данных.

EF6 Вычисления на стороне клиента – это новая и полезная функция в EF Core.

Вычисление на стороне клиента дает возможность адаптировать или изменять данные в последней части запроса, что может избавить вас от необходимости применять дополнительный этап обработки после запроса. В разделе 2.6 мы используем эту функциональность, чтобы создать список авторов книги, разделенных запятыми. Если не использовать вычисление на стороне клиента для этой задачи, то нужно будет (а) отправить обратно список всех имен авторов и (б) добавить дополнительный этап обработки после запроса, используя `foreach`, чтобы применить команду `string.Join` к авторам каждой книги.

Предупреждение: EF Core выбросит исключение, если не сможет преобразовать ваш LINQ

До EF Core 3 любой код LINQ, который нельзя было преобразовать в команду базы данных, выполнялся в программе с использованием вычисления на стороне клиента. В некоторых случаях такой подход может приводить к крайне неэффективным запросам. (Я писал об этом в первом издании.) EF Core 3 изменил эту ситуацию, чтобы это вычисление использовалось только на последнем этапе LINQ-запросов, не позволяя производить неэффективные запросы.

Однако это изменение создает другую проблему: если LINQ-запросы нельзя преобразовать в команды базы данных, то EF Core выбросит исключение `InvalidOperationException` с сообщением `could not be translated`. Проблема состоит в том, что эту ошибку можно получить только при попытке выполнить данный запрос – а вы не хотите, чтобы эта ошибка произошла в промышленном окружении!

В этой книге я буду помогать вам писать запросы, которые будут работать, но при работе со сложными запросами можно легко получить не совсем правильное LINQ-выражение, что приведет к возникновению исключения `InvalidOperationException`. Такое иногда происходит и у меня, хотя я хорошо знаю EF Core. Поэтому в главе 17 я рекомендую провести модульное тестирование доступа к базе данных с реальной базой данных и/или иметь набор интеграционных тестов.

Для отображения списка книг в приложении Book App необходимо (а) извлечь все имена авторов по порядку из таблицы Authors и (б) объединить их в одну строку с запятыми между именами. Вот пример загрузки двух свойств: `BookId` и `Title`, обычным способом, и третьего свойства, `AuthorsString`, с использованием вычисления на стороне клиента.

Листинг 2.10 Запрос с методом `Select`, включающий в себя не относящуюся к SQL команду `string.Join`

```
var firstBook = context.Books
    .Select(book => new
    {
```

```
        book.BookId,
        book.Title,
        AuthorsString = string.Join(", ",
            book.AuthorsLink
            .OrderBy(ba => ba.Order)
            .Select(ba => ba.Author.Name))
    }
    ).First();
```

Команда string.Join выполняется на клиенте в приложении

Эти части можно преобразовать в SQL и запустить на сервере

Выполнение этого кода для книги, у которой два автора, Джек и Джилл, приведет к тому, что `AuthorsString` будет содержать строку «Джек, Джилл», а `BookId` и `Title` будут иметь значение соответствующих столбцов в таблице Books. На рис. 2.9 показано, что листинг 2.10 будет обрабатываться в четыре этапа. Я хочу сосредоточиться на этапе 3, где EF Core выполняет код, который не удалось преобразовать в SQL, на стороне клиента.

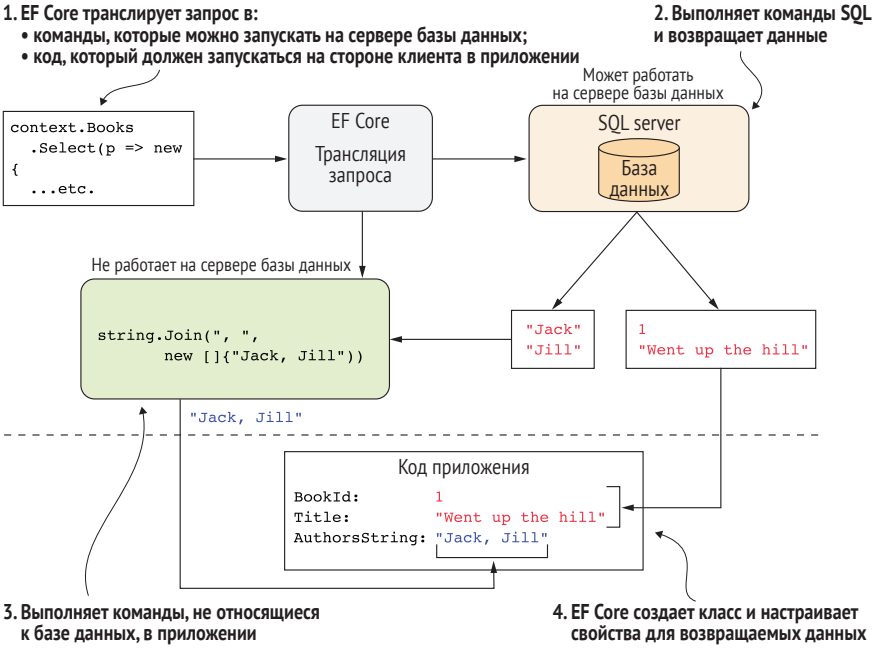


Рис. 2.9 Некоторые части запроса преобразуются в SQL и выполняются в SQL Server; оставшаяся часть, string.Join, должна выполняться EF Core на стороне клиента до того, как объединенный результат вернется в код приложения

Пример из листинга 2.10 довольно простой, но нужно быть осторожным при использовании свойства, создаваемого вычислением на стороне клиента. Использование этого вычисления для свойства означает, что вы не можете использовать это свойство в какой бы то ни было команде LINQ, которая будет создавать запросы базе данных, например для сортировки или фильтрации этого свойства. Если

вы это сделаете, то получите исключение `InvalidOperationException` с сообщением `could not be translated`. Например, на рис. 2.9 если вы попытаетесь отсортировать или отфильтровать `AuthorsString`, то получите исключение `could not be translated`.

2.6 Создание сложных запросов

Изучив основы выполнения запросов к базе данных, рассмотрим примеры, которые чаще встречаются в реальных приложениях. Мы создадим запрос, который перечисляет все книги в приложении Book App, используя ряд функциональных возможностей, включая сортировку, фильтрацию и разбиение на страницы.

Можно создать отображение книги, используя немедленную загрузку. Сначала вы загружаете все данные; затем объединяете авторов, рассчитываете стоимость, подсчитываете средний рейтинг и т. д. Проблема с этим подходом заключается в том, что, во-первых, вы загружаете данные, которые вам не нужны, а во-вторых, сортировку и фильтрацию необходимо выполнять внутри вашего приложения. Для приложения Book App из этой главы, которое насчитывает около 50 книг, конечно, можно загрузить все книги и связи в память, а затем отсортировать или отфильтровать их в приложении, но такой подход не сработает для Amazon!

Более подходящее решение – вычислить значения внутри SQL Server, чтобы можно было выполнить сортировку и фильтрацию до того, как данные вернутся в приложение. В оставшейся части этой главы мы будем использовать выборочную загрузку, которая объединяет выбор, сортировку, фильтрацию и разбиение по страницам в один большой запрос. В этом разделе мы начнем с выбора. Однако, прежде чем продемонстрировать выборочный запрос, который загружает данные книги, вернемся к отображению информации по книге «*Quantum Networking*» из начала главы. На этот раз на рис. 2.10 показаны отдельные LINQ-запросы, необходимые для получения каждого фрагмента данных.

Это трудный для понимания рисунок, потому что запросы, необходимые для получения всех данных, сложные. Учитывая эту диаграмму, посмотрим, как создать запрос для выбора книги. Начнем с класса, в который вы собираетесь поместить данные. Этот тип класса, который существует только для того, чтобы собрать воедино нужные вам данные, называется по-разному. В ASP.NET это ViewModel, но данный термин также имеет другие значения и варианты использования; поэтому я называю этот тип класса *Data Transfer Object* (*DTO*). В листинге 2.11 показан DTO-класс `BookListDto`.

ОПРЕДЕЛЕНИЕ Существует множество определений объекта передачи данных (DTO), но то, которое подходит для мое-

го использования DTO, звучит так: это «объект, используемый для инкапсуляции данных и их отправки из одной подсистемы приложения в другую» (Stack Overflow, https://stackoverflow.com/a/1058186/1434764).

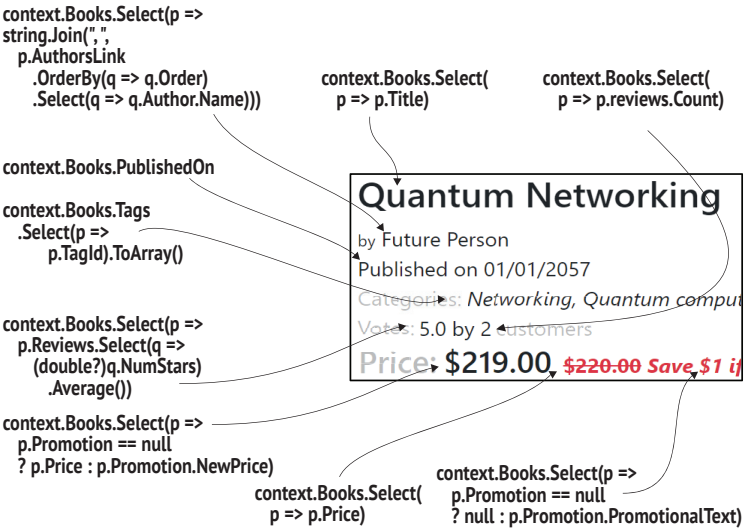


Рис. 2.10 Отдельные запросы, необходимые для построения отображения списка книг, со всеми частями запроса, используемыми для предоставления значения, необходимого для этой части отображения. Некоторые запросы простые, например получить название книги, но другие не так очевидны, например вычисление средней оценки по отзывам

Листинг 2.11 Класс BookListDto

```
public class BookListDto
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
    public decimal Price { get; set; }
    public decimal
        ActualPrice { get; set; }
    public string
        PromotionPromotionalText { get; set; }
    public string AuthorsOrdered { get; set; }

    public int ReviewsCount { get; set; }
    public double?
        ReviewsAverageVotes { get; set; }
    public string[] TagStrings { get; set; }
}
```

Вам понадобится первичный ключ, если клиент нажимает на запись, чтобы купить книгу

Хотя дата публикации не отображается, нужно будет выполнить сортировку по ней, поэтому надо включить ее

Обычная цена книги

Цена – обычная или promotional.NewPrice, если есть

Рекламный текст, показывающий, есть ли новая цена

Среднее значение всех голосов либо null, если голосов нет

Строка для хранения списка имен авторов, разделенных запятыми

Количество людей, оставивших отзывы по книге

Теги (т. е. категории) для этой книги

Для работы с выборочной загрузкой у класса, который будет получать данные, должен быть конструктор по умолчанию (конструктор без параметров), класс не должен быть статическим, а свойства должны иметь открытые методы записи.

Затем мы создадим запрос на выборку, который заполнит все свойства в `BookListDto`. Поскольку нам нужно использовать этот запрос с другими частями запроса, такими как сортировка, фильтрация и разбиение на страницы, то мы воспользуемся типом `IQueryable<T>` для создания метода `MapBookToDto`, который принимает `IQueryable<Book>` и возвращает `IQueryable<BookListDto>`. Этот метод показан в следующем листинге. Как видно, `Select` объединяет все отдельные запросы, которые вы видели на рис. 2.10.

Листинг 2.12 Запрос с `Select` для заполнения `BookListDto`

```
public static IQueryable<BookListDto>
    MapBookToDto(this IQueryable<Book> books)
{
    return books.Select(book => new BookListDto
    {
        BookId = book.BookId,
        Title = book.Title,
        Price = book.Price,
        PublishedOn = book.PublishedOn,
        ActualPrice = book.Promotion == null
            ? book.Price
            : book.Promotion.NewPrice,
        PromotionPromotionalText =
            book.Promotion == null
                ? null
                : book.Promotion.PromotionalText,
        AuthorsOrdered = string.Join(", ",
            book.AuthorsLink
                .OrderBy(ba => ba.Order)
                .Select(ba => ba.Author.Name)),
        ReviewsCount = book.Reviews.Count,
        ReviewsAverageVotes =
            book.Reviews.Select(review =>
                (double?) review.NumStars).Average(),
        TagStrings = book.Tags
            .Select(x => x.TagId).ToArray(),
    });
}
```

Принимает IQueryable<Book> и возвращает IQueryable<BookListDto>

Простые копии существующих столбцов в таблице Book

Вычисляет обычную цену или цену по рекламной акции, если такая связь существует

PromotionalText зависит от того, есть ли PriceOffer для этой книги

Получает массив имен авторов в правильном порядке. Мы используем вычисления на стороне клиента, потому что хотим, чтобы имена авторов были объединены в одну строку

Нужно рассчитать количество отзывов

Чтобы EF Core превратил Average в SQL-команду AVG, нужно преобразовать NumStars в (double?)

Массив тегов (категорий) для этой книги

ПРИМЕЧАНИЕ Отдельные части выборочного запроса из листинга 2.12 представляют собой повторяющийся код, который я упоминал, рассказывая о моменте озарения в главе 1. Глава 6 познакомит вас с инструментами отображения, которые автоматизируют бóльшую часть написания этого кода, но в первой

части книги я даю весь код полностью, чтобы вы видели всю картину. Будьте уверены, что есть способ автоматизировать выполнение запросов, где используется метод `Select`, и это повысит вашу продуктивность.

Метод `MapBookToDto` использует паттерн «Объект–запрос»; он принимает `IQueryable<T>` и возвращает `IQueryable<T>`, что позволяет инкапсулировать запрос или часть запроса в методе. Таким образом, запрос изолирован в одном месте, что упрощает поиск, отладку и настройку производительности. Мы также будем использовать этот паттерн для сортировки, фильтрации и разбиения на страницы.

ПРИМЕЧАНИЕ Данный паттерн полезен при построении таких запросов, как вывод списка книг в этом примере, но существуют и альтернативные подходы, например паттерн «Репозиторий».

Метод `MapBookToDto` в .NET называется *методом расширения*. Эти методы позволяют объединять объекты запроса в цепочку. Вы увидите, как используется эта цепочка, в разделе 2.9, когда мы объединим каждую часть запроса на вывод списка книг для создания окончательного, составного запроса.

ПРИМЕЧАНИЕ Метод может стать методом расширения, если (а) он объявлен в статическом классе, (б) метод является статическим и (в) перед первым параметром идет ключевое слово `this`.

Объекты запроса принимают на входе `IQueryable<T1>` и возвращают `IQueryable<T2>`, поэтому получается, что мы добавляем команды LINQ в `IQueryable<T1>`. Можно добавить еще один объект запроса в конец или, если вы хотите выполнить запрос, добавить команду выполнения (см. рис. 2.8), например `ToList`, для выполнения запроса. Вы увидите этот подход в действии в разделе 2.9, когда мы объединим объекты запроса выбора, сортировки, фильтрации и разбиения на страницы, которые EF Core превращает в довольно эффективный запрос к базе данных. В главе 15 мы проработаем серию настроек производительности, чтобы сделать запрос на получение списка книг еще быстрее.

ПРИМЕЧАНИЕ Результаты этого запроса можно увидеть, клонировав код из репозитория Git, а затем запустив приложение Book App локально. Меню **Logs** (Журналы) покажет вам SQL, используемый для загрузки списка книг с выбранными вами параметрами сортировки, фильтрации и разбиения по страницам.

2.7 Знакомство с архитектурой приложения Book App

Я дождался момента, чтобы поговорить о дизайне приложения Book App, потому что теперь, когда мы создали класс `BookListDto`, это должно иметь больше смысла. На данном этапе у нас есть классы сущностей (`Book`, `Author` и т. д.), которые отображаются в базу данных с помощью EF Core. Также у нас есть класс `BookListDto`, который содержит данные в той форме, которая нужна стороне представления – в данном случае веб-серверу ASP.NET Core.

В простом приложении можно поместить классы сущностей в одну папку, DTO в другую и т. д. Но даже в небольшом приложении, таком как приложение Book App, подобная практика может сбивать с толку, потому что подход, который вы используете с базой данных, отличается от подхода, который вы используете при отображении данных клиенту. Принцип разделения ответственности (http://mng.bz/7Vom) гласит, что программное обеспечение должно быть разбито на отдельные части. Например, запрос к базе данных для отображения книг не должен содержать код, создающий HTML-код для показа книг пользователю.

Можно разделить части приложения Book App разными способами, но мы будем использовать распространенный дизайн, который называется *многослойной архитектурой*. Этот подход хорошо работает для приложений .NET малого и среднего размера. На рис. 2.11 показана архитектура приложения Book App из этой главы.

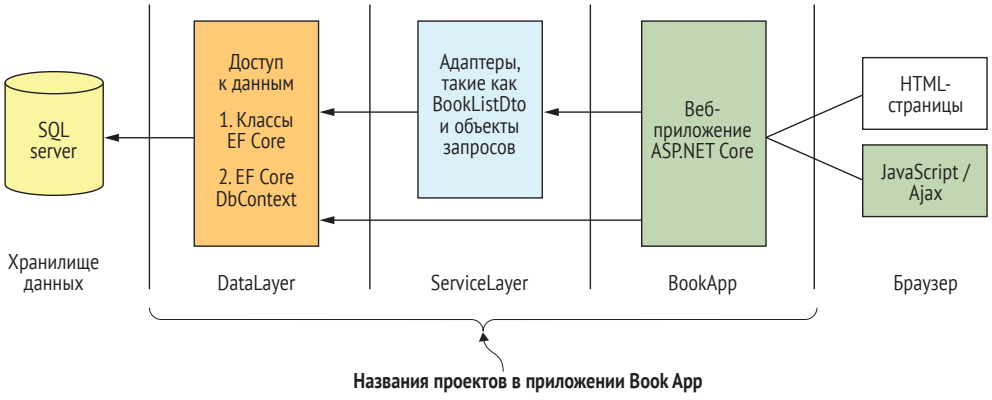


Рис. 2.11 Многослойная архитектура для приложения Book App. Размещение частей кода в отдельных проектах позволяет четче обозначить функционал каждой части. DataLayer, например, должен беспокоиться только о базе данных и не должен знать, как данные будут использоваться, – принцип разделения ответственности в действии. Стрелки всегда указывают влево, потому что нижестоящие (слева) проекты не могут получить доступ к вышестоящим (справа) проектам

Три больших прямоугольника – это проекты .NET, имена которых указаны внизу. Классы и код этих трех проектов разделены следующим образом:

- *DataLayer* – основное внимание в этом слое уделяется доступу к базе данных. В этом проекте находятся классы сущностей и DbContext приложения. Этот слой ничего не знает о слоях, которые находятся над ним;
- *ServiceLayer* – этот слой действует как адаптер между DataLayer и веб-приложением ASP.NET Core, используя DTO, объекты запроса и различные классы для выполнения команд. Идея состоит в том, что слой представления ASP.NET Core должен столько всего сделать, что ServiceLayer передает данные, подготовленные для отображения;
- *BookApp* – иначе *слой представления* – ориентирован на представление данных в удобном для пользователя виде. Слой представления должен фокусироваться только на взаимодействии с пользователем, поэтому по возможности мы убираем отсюда как можно больше вещей, касающихся базы данных и адаптации данных. В приложении Book App мы будем использовать веб-приложение ASP.NET Core, обслуживающее в основном HTML-страницы с небольшим количеством JavaScript, работающего в браузере.

Использование многослойной архитектуры делает приложение Book App немного более сложным для понимания, но это один из способов создания реальных приложений. Использование слоев также облегчает понимание предназначения того или иного участка кода, потому что код разных слоев не смешивается.

2.8 Добавляем сортировку, фильтрацию и разбиение на страницы

Разобравшись со структурой проекта, можно быстрее продолжить работу и создать оставшиеся объекты запроса для окончательного отображения списка книг. Я начну с того, что покажу вам скриншот (рис. 2.12) с элементами управления сортировкой, фильтрацией и разбиением на страницы, чтобы у вас было представление о том, что вы реализуете.

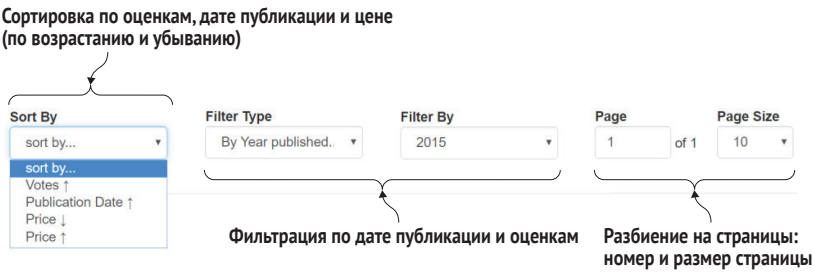


Рис. 2.12 Три команды – сортировка, фильтрация и разбиение по страницам, – как показано на домашней странице приложения Book App. Можно увидеть эту страницу в действии, если запустить приложение из прилагаемого репозитория Git

2.8.1 Сортировка книг по цене, дате публикации и оценкам покупателей

Сортировка в LINQ выполняется методами `OrderBy` и `OrderByDescending`. В качестве метода расширения создается объект запроса `OrderBooksBy`, как показано в следующем листинге. Вы увидите, что помимо параметра `IQueryable<BookListDto>` этот метод принимает параметр `enum`, определяющий нужный пользователю тип сортировки.

Листинг 2.13 Метод `OrderBooksBy`

```
public static IQueryable<BookListDto> OrderBooksBy
    (this IQueryable<BookListDto> books,
     OrderByOptions orderByOptions)
{
    switch (orderByOptions)
    {
        case OrderByOptions.SimpleOrder:
            return books.OrderByDescending(
                x => x.BookId);
        case OrderByOptions.ByVotes:
            return books.OrderByDescending(x =>
                x.ReviewsAverageVotes);
        case OrderByOptions.ByPublicationDate:
            return books.OrderByDescending(
                x => x.PublishedOn);
        case OrderByOptions.ByPriceLowestFirst:
            return books.OrderBy(x => x.ActualPrice);
        case OrderByOptions.ByPriceHighestFirst:
            return books.OrderByDescending(
                x => x.ActualPrice);
            default:
                throw new ArgumentOutOfRangeException(
                    nameof(orderByOptions), orderByOptions, null);
    }
}
```

Из-за разбиения на страницы всегда нужно выполнять сортировку. По умолчанию сортировка выполняется по первичному ключу. Это делается быстро

Упорядочение книг в зависимости от голосов. Книги, у которых нет ни одного голоса (возвращается null), идут внизу

Упорядочение по дате публикации, вверху – самые свежие книги

Упорядочение по актуальной цене с учетом рекламной цены (по возрастанию и по убыванию цены)

Вызов метода `OrderBooksBy` возвращает исходный запрос с соответствующей командой сортировки LINQ, добавленной в конце. Этот запрос передается следующему объекту запроса, или если вы закончили, то вызывается команда для получения результата, например `ToList`.

ПРИМЕЧАНИЕ Даже если пользователь не выберет сортировку, она все равно будет выполнена (см. оператор `SimpleOrder`), потому что вы будете использовать разбиение на страницы, предоставляя только одну страницу за раз, вместо всех данных, а SQL обязательно требуется сортировка данных для корректного разбиения на страницы. Наиболее эффективная сортировка выполняется по первичному ключу, поэтому по умолчанию вы будете использовать его.

2.8.2 Фильтрация книг по году публикации, категориям и оценкам покупателей

Фильтрация, созданная для приложения Book App, немного сложнее по сравнению с сортировкой, рассмотренной в разделе 2.8.1, потому что сначала мы заставляем покупателя выбрать нужный ему тип фильтра, а затем выбрать фактическое значение фильтра. Значение фильтра для `Votes` простое: это набор фиксированных значений (4 или выше, 3 или выше и т. д.), а для категории это `TagId`. Но для фильтрации по дате нужно найти даты публикаций, чтобы поместить их в выпадающий список.

Будет поучительно взглянуть на код для определения дат выпуска книг, потому что этот код – хороший пример объединения команд LINQ для создания окончательного выпадающего списка. Вот фрагмент кода из метода `GetFilterDropDownValues`.

Листинг 2.14 Код для создания списка дат выпуска книг

```
var result = _db.Books
    .Where(x => x.PublishedOn <= DateTime.UtcNow.Date)
    .Select(x => x.PublishedOn.Year)
    .Distinct()
    .OrderByDescending(x => x)
    .Select(x => new DropdownTuple
    {
        Value = x.ToString(),
        Text = x.ToString()
    }).ToList();
var comingSoon = _db.Books.
    Any(x => x.PublishedOn > DateTime.Today);
if (comingSoon)
    result.Insert(0, new DropdownTuple
    {
        Value = BookListDtoFilter.AllBooksNotPublishedString,
        Text = BookListDtoFilter.AllBooksNotPublishedString
    });

return result;
```

Загружает книги, отфильтровывая будущие книги; затем выбирает годы, когда эти книги были опубликованы

Метод Distinct убирает дубликаты из списка годов публикации

Упорядочение по годам, наверху – самый последний год

Наконец, я использую два вычисления на стороне клиента, чтобы преобразовать значения в строки

Возвращает true, если книга из списка еще не опубликована

Добавляет фильтр «готовятся к выходу» для всех будущих книг

Результат этого кода – список пар `Value`/`Text`, где хранится год публикации, а также раздел «Готовятся к выходу» для книг, которые еще не опубликованы. Эти данные превращаются в раскрывающийся HTML-список с помощью ASP.NET Core и отправляются в браузер.

В следующем листинге показан объект запроса, `FilterBooksBy`, принимающий в качестве входных данных часть `Value` раскрывающегося списка, созданного в листинге 2.14, плюс тип фильтрации, запрошенный покупателем.

Листинг 2.15 Метод `FilterBooksBy`

```
public static IQueryable<BookListDto> FilterBooksBy(
    this IQueryable<BookListDto> books,
    BooksFilterBy filterBy, string filterValue)
{
    if (string.IsNullOrEmpty(filterValue))
        return books;

    switch (filterBy)
    {
        case BooksFilterBy.NoFilter:
            return books;
        case BooksFilterBy.ByVotes:
            var filterVote = int.Parse(filterValue);
            return books.Where(x =>
                x.ReviewsAverageVotes > filterVote);
        case BooksFilterBy.ByTags:
            return books.Where(x => x.TagStrings
                .Any(y => y == filterValue));
        case BooksFilterBy.ByPublicationYear:
            if (filterValue == AllBooksNotPublishedString)
                return books.Where(
                    x => x.PublishedOn > DateTime.UtcNow);
            var filterYear = int.Parse(filterValue);
            return books.Where(
                x => x.PublishedOn.Year == filterYear
                     && x.PublishedOn <= DateTime.UtcNow);
        default:
            throw new ArgumentOutOfRangeException
                (nameof(filterBy), filterBy, null);
    }
}
```

В метод передается тип фильтра и значение фильтра, выбранное пользователем

Если значение фильтра не задано, возвращаем IQueryable без изменений

Если фильтр не выбран, возвращаем IQueryable без изменений

Фильтр по голосам возвращает только книги со средней оценкой выше значения filterVote. Если на книгу нет отзывов, то свойство ReviewsAverageVotes будет иметь значение null, а проверка всегда возвращает false

Выбирает любые книги с категорией Tag, соответствующей filterValue

Если был выбран вариант «Готовятся к выходу», возвращаются только те книги, которые еще не опубликованы

Если у нас есть конкретный год, то фильтруем по нему. Обратите внимание, что мы также удаляем будущие книги (если пользователь выбрал дату в этом году)

2.8.3 Другие параметры фильтрации: поиск текста по определенной строке

Можно было бы создать множество других типов фильтров / поисков по книгам, и очевидный кандидат – это поиск по названию. Но нужно убедиться, что команды LINQ, которые вы используете для поиска в строке, выполняются в базе данных, потому что это будет работать намного лучше загрузки всех данных и фильтрации их внутри приложения. EF Core преобразует следующий код C# в LINQ-запросе в команду базы данных: `==`, `Equal`, `StartsWith`, `EndsWith`, `Contains` и `IndexOf`. В табл. 2.1 показаны некоторые из этих команд в действии.

Еще одна важная вещь, которую нужно знать: чувствительность к регистру строкового поиска, выполняемого с помощью команд SQL, зависит от типа базы данных, а в некоторых базах данных это правило называется *сопоставлением* (*collation*). По умолчанию в базе данных

SQL Server используется сопоставление без учета регистра, поэтому поиск `Cat` приведет к поиску `cat` и `Cat`. Множество баз данных SQL по умолчанию нечувствительны к регистру, но в Sqlite поиск можно вести с учетом регистра и без него (см. модульный тест `Ch02_StringSearch` из репозитория для получения более подробной информации), а в Cosmos DB поиск ведется с учетом регистра по умолчанию.

Таблица 2.1 Пример строковых команд .NET в базе данных SQL Server

Строковая команда	Пример (находит заголовок со строкой "The Cat sat on the mat.")

`StartsWith`	`var books = context.Books`
	`  .Where(p => p.Title.StartsWith("The"))`
	`  .ToList();`

`EndsWith`	`var books = context.Books`
	`  .Where(p => p.Title.EndsWith("MAT."))`
	`  .ToList();`

`Contains`	`var books = context.Books` `  .Where(p => p.Title.Contains("cat"))`

EF Core 5 предоставляет различные способы настройки сопоставления в базе данных. Обычно сопоставление настраивают для всей базы данных или определенного столбца (описано в разделе 7.7), но также можно определить сопоставление в запросе с помощью метода `EF.Functions.Collate`. Следующий фрагмент кода устанавливает сопоставление SQL Server. Это означает, что данный запрос будет сравнивать строку, используя сопоставление `Latin1_General_CS_AS` (с учетом регистра):

```
context.Books.Where( x =>
      EF.Functions.Collate(x.Title, "Latin1_General_CS_AS")
          == "HELP" //This does not match "help"
```

ПРИМЕЧАНИЕ Определение того, что является верхним регистром в большом количестве языков со множеством видов письма, – сложная проблема! К счастью, реляционные базы данных выполняют эту задачу на протяжении многих лет, а в SQL Server существует более 200 сопоставлений.

Еще одна строковая команда – это SQL-команда `LIKE`, к которой можно получить доступ через метод `EF.Function.Like`. Она обеспечивает простое сопоставление с образцом, используя _ (нижнее подчеркивание) для сопоставления одного символа и знак % для сопоставления любого количества символов.

Следующий фрагмент кода соответствует фразе `"The Cat sat on the mat."` и `"The dog sat on the step."`, но не `"The rabbit sat on the hutch."`, потому что длина слова `rabbit` больше трех букв:

```
var books = context.Books
    .Where(p => EF.Functions.Like(p.Title, "The ___ sat on the %."))
    .ToList();
```

ДРУГИЕ ВАРИАНТЫ ЗАПРОСА: СЛОЖНЫЕ ЗАПРОСЫ (GROUPBY, SUM, MAX И Т. Д.)

В этой главе рассматривается широкий спектр команд запроса, но EF Core может транслировать гораздо больше команд в большинство баз данных. В разделе 6.1.8 описаны команды, требующие пояснения или написания особого кода.

2.8.4 Разбиение книг на страницы в списке

Если вы хоть раз использовали поиск Google, это значит, что вы использовали разбиение по страницам. Google представляет первые десять или вроде того результатов, а остальные результаты можно просмотреть с помощью перехода на последующие страницы. Наше приложение Book App использует разбиение на страницы, которое очень просто реализовать с помощью методов `Skip` и `Take`.

Хотя другие объекты запроса были привязаны к классу `BookListDto`, потому что LINQ-команды разбиения на страницы очень просты, можно создать универсальный объект запроса, который будет работать с любым запросом `IQueryable<T>`. Этот объект показан в следующем листинге. Он использует получение номера страницы в правильном диапазоне, но другая часть приложения должна делать это в любом случае, чтобы отображать правильную информацию о страницах на экране.

Листинг 2.16 Универсальный метод Page

```
public static IQueryable<T> Page<T>(
    this IQueryable<T> query,
    int pageNumZeroStart, int pageSize)
{
    if (pageSize == 0)
        throw new ArgumentOutOfRangeException
            (nameof(pageSize), "pageSize cannot be zero.");

    if (pageNumZeroStart != 0)
        query = query
            .Skip(pageNumZeroStart * pageSize);

    return query.Take(pageSize);
}
```

Пропускает правильное количество страниц

Принимает число для размера текущей страницы

Как я сказал ранее, разбиение на страницы работает только в том случае, если данные упорядочены. В противном случае SQL Server сгенерирует исключение, потому что реляционные базы данных не гарантируют порядок, в котором возвращаются данные; в реляционной базе данных нет порядка строк по умолчанию.

2.9 Собираем все вместе: объединение объектов запроса

Мы рассмотрели каждый объект запроса, необходимый для создания списка книг для приложения Book App. Теперь пора посмотреть, как объединить эти объекты для создания составного запроса, чтобы работать с сайтом. Преимущество построения сложного запроса из отдельных частей заключается в том, что такой подход упрощает написание и тестирование общего запроса, потому что можно протестировать каждую часть по отдельности.

В листинге 2.17 показан класс `ListBooksService`, у которого есть один метод `SortFilterPage`, использующий все объекты запроса (выбор, сортировка, фильтрование и разбиение на страницы) для создания составного запроса. Также ему необходим DbContext приложения для доступа к свойству `Books`, которое предоставляется через конструктор.

> **СОВЕТ** В листинге 2.17 жирным шрифтом выделен метод `AsNoTracking`. Этот метод не позволяет EF Core делать снимок отслеживания (см. рис. 1.6) для запросов с доступом только на чтение, что делает запрос немного быстрее. Вы должны использовать его в любых запросах с доступом только на чтение (в которых вы читаете данные, но никогда не обновляете их). В данном случае мы не загружаем никакие классы сущностей, поэтому это лишнее. Но я поместил его туда, чтобы напомнить, что это запрос с доступом только на чтение.

Листинг 2.17 Класс `ListBookService`, предоставляющий отсортированный, отфильтрованный и постраничный список

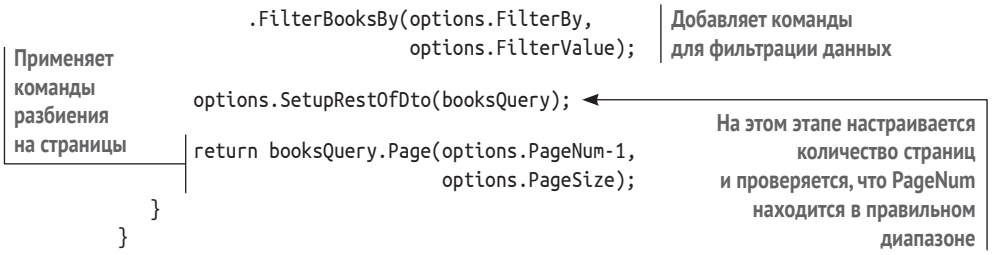
```
public class ListBooksService
{
    private readonly EfCoreContext _context;

    public ListBooksService(EfCoreContext context)
    {
        _context = context;
    }

    public IQueryable<BookListDto> SortFilterPage
        (SortFilterPageOptions options)
    {
        var booksQuery = _context.Books
            .AsNoTracking()
            .MapBookToDto()
            .OrderBooksBy(options.OrderByOptions)
```

Начинает с выбора свойства Books в DbContext приложения

Поскольку этот запрос с доступом только на чтение, добавляем метод .AsNoTracking

Использует объект запроса выбора, который выбирает/вычисляет нужные ему данные

Добавляет команды для упорядочивания данных с использованием заданных параметров



Как видите, четыре объекта запроса – выбор, сортировка, фильтрация и разбиение на страницы – добавляются по очереди (это называется *цепочкой*) для формирования окончательного составного запроса. Обратите внимание, что код `options.SetupRestOfDto(booksQuery)` перед объектом запроса разбиения на страницы отвечает за такие вещи, как количество страниц, гарантию, что `PageNum` находится в правильном диапазоне, и выполнение нескольких других вспомогательных операций. В главе 5 показано, как вызывается `ListBooksService` в веб-приложении ASP.NET Core.

Резюме

- Чтобы получить доступ к базе данных любым способом через EF Core, необходимо определить DbContext приложения.
- Запрос EF Core состоит из трех частей: свойство приложения DbContext, серия команд LINQ / EF Core и команда для выполнения запроса.
- Используя EF Core, можно смоделировать три основные связи в базе данных: «один к одному», «один ко многим» и «многие ко многим». Другие связи рассматриваются в главе 8.
- Классы, которые EF Core отображает в базу данных, называются *классами сущностей*. Я использую этот термин, чтобы подчеркнуть тот факт, что класс, о котором я говорю, отображается EF Core в базу данных.
- Если вы загружаете класс сущности, по умолчанию он не загружает ни одну из своих связей. Например, при запросе класса сущности `Book` его свойства связей загружены не будут (`Reviews`, `AuthorsLink` и `Promotion`); они останутся равны `null`.
- Можно загрузить связанные данные, прикрепленные к классу сущности, четырьмя способами: немедленная загрузка, явная загрузка, выборочная загрузка и отложенная загрузка.
- Вычисление на стороне клиента позволяет последнему этапу запроса содержать методы, такие как `string.Join`, которые нельзя преобразовать в SQL-команды.
- Я использую термин *объект запроса* для обозначения инкапсулированного запроса или части запроса. Эти объекты часто создаются

как методы расширения .NET, а это означает, что их можно легко связать в цепочку, подобно тому, как пишется LINQ.
- Выбор, сортировка, фильтрация и разбиение по страницам – это обычные запросы, которые можно инкапсулировать в объект запроса.
- Если вы внимательно пишете свои LINQ-запросы, то можете переместить агрегированные вычисления, такие как `Count`, `Sum` и `Average`, в реляционную базу данных, улучшая производительность.

Для читателей, знакомых с EF6.x:
- многие концепции, описанные в этой главе, такие же, как и в EF6.x. В некоторых случаях (например, немедленная загрузка) команды и/или конфигурация EF Core незначительно меняются, но часто в лучшую сторону.

3 Изменение содержимого базы данных

В этой главе рассматриваются следующие темы:

- создание новой строки в таблице базы данных;
- обновление существующих строк в таблице базы данных для двух типов приложений;
- обновление сущностей со связями «один к одному», «один ко многим» и «многие ко многим»;
- удаление отдельных сущностей и сущностей со связями из базы данных.

В главе 2 мы рассмотрели, как выполнять запросы к базе данных. В этой главе мы переходим к изменению содержимого базы данных. Изменение данных состоит из трех частей: создание новых строк в таблице базы данных, обновление существующих строк и их удаление – и я буду рассказывать о них именно в таком порядке. *Create* (Создание), *Update* (Обновление) и *Delete* (Удаление) наряду с *Read* (Чтение) (что согласно терминологии EF Core является запросом) – это понятия базы данных, описывающие происходящее в ней. Часто вместо этих четырех слов используется их аббревиатура – *CRUD*.

Мы будем использовать ту же базу данных, что и в главе 2, которая содержит классы сущностей `Book`, `PriceOffer`, `Review`, `BookAuthor` и `Author`. Эти классы обеспечивают хороший набор типов свойств и связей, которые можно использовать для изучения различных проблем и подходов к изменению данных в базе данных через EF Core.

3.1 Представляем свойство сущности State

Прежде чем приступить к описанию методов добавления, обновления или удаления сущностей, хочу познакомить вас со свойством сущности EF Core `State`. Это свойство обеспечивает еще один взгляд на то, как работает EF Core, что поможет вам понять, что происходит, когда вы добавляете, обновляете или удаляете сущности.

Любой экземпляр класса сущности имеет состояние (`State`), к которому можно получить доступ, используя следующую команду: `context.Entry(someEntityInstance).State`. `State` сообщает EF Core, что делать с этим экземпляром при вызове метода `SaveChanges`. Вот список возможных состояний и того, что происходит, если вызвать `SaveChanges`:

- `Added` – сущность необходимо создать в базе данных. Метод `SaveChanges` вставляет ее;
- `Unchanged` – сущность находится в базе данных и не была изменена. Метод `SaveChanges` игнорирует ее;
- `Modified` – сущность находится в базе данных и была изменена. Метод `SaveChanges` обновляет ее;
- `Deleted` – сущность находится в базе данных, но ее нужно удалить. Метод `SaveChanges` удаляет ее;
- `Detached` – сущность не отслеживается. Метод `SaveChanges` ее не видит.

Обычно вы не смотрите и не изменяете состояние напрямую. Вы используете различные команды, перечисленные в этой главе, чтобы добавлять, обновлять или удалять сущности. Эти команды гарантируют, что состояние устанавливается в *отслеживаемой сущности* (см. определение ниже). При вызове метода `SaveChanges` он смотрит на все отслеживаемые сущности и их состояние, чтобы решить, какие изменения необходимо применить к базе данных. В оставшейся части главы я ссылаюсь на свойство `State` сущности, чтобы показать, как EF Core решает, какой тип изменения применить к базе данных.

> **ОПРЕДЕЛЕНИЕ** *Отслеживаемые сущности* – это экземпляры сущностей, считанные из базы данных с помощью запроса, который не включал вызов метода `AsNoTracking`. Как вариант, после того как экземпляр сущности был использован в качестве параметра для методов EF Core (например, `Add`, `Update` или `Delete`), он становится отслеживаемым.

3.2 Создание новых строк в таблице

Создание новых данных в базе данных – это добавление (с помощью SQL-команды `INSERT` в реляционной базе данных) новой строки в таблицу. Если, например, вы хотите добавить нового автора в при-

ложение Book App, это будет называться операцией создания в базе данных.

С точки зрения EF Core, создание новых данных в базе данных является самой простой операцией обновления, поскольку EF Core может принимать набор связанных классов сущностей, сохранять их в базу данных и самостоятельно разбираться с внешними ключами, необходимыми для связывания. В этом разделе мы начнем с простого примера, а затем перейдем к более сложным.

3.2.1 Самостоятельное создание отдельной сущности

Начнем с класса сущности, у которого нет навигационных свойств, то есть связей с другими таблицами в базе данных. Это редкий пример, но он показывает два этапа операции создания:

1. добавление сущности в DbContext приложения;
2. вызов метода `SaveChanges`.

В этом листинге мы создаем экземпляр класса сущности `ExampleEntity` и добавляем новую строку в таблицу. Созданный экземпляр в данном случае сопоставляется с таблицей ExampleEntities.

Листинг 3.1 Пример создания отдельной сущности

```
var itemToAdd = new ExampleEntity
{
    MyMessage = "Hello World"
};
context.Add(itemToAdd);
context.SaveChanges();
```

Использует метод Add для добавления ExampleEntity в DbContext приложения. DbContext определяет таблицу, в которую нужно ее добавить, в зависимости от типа параметра

Вызывает метод SaveChanges из DbContext приложения для обновления базы данных

Поскольку вы добавляете экземпляр сущности `itemToAdd`, которая изначально не отслеживалась, EF Core начинает отслеживать ее и устанавливает для ее свойства `State` значение `Added`. После вызова метода `SaveChanges` EF Core находит отслеживаемую сущность типа `ExampleEntity` с состоянием `Added`, поэтому она добавляется как новая строка в таблицу базы данных, связанную с классом `ExampleEntity`.

> **EF6** В EF6.x нужно было добавить `itemToAdd` в свойство `DbSet<ExampleEntity>` в DbContext приложения, например `context.ExampleEntities.Add(itemToAdd)`. Данный подход все еще актуален, но EF Core представила сокращенный вариант записи, показанный в листинге 3.1, который применяется к методам `Add`, `Remove`, `Update` и `Attach` (см. главу 11 для получения дополнительной информации о последних двух методах). EF Core определяет, какую сущность вы изменяете, глядя на тип предоставляемого вами экземпляра.

EF Core создает SQL-команду для обновления базы данных SQL Server.

Листинг 3.2 Команды SQL, созданные для вставки новой строки в таблицу ExampleEntities

```
SET NOCOUNT ON;
INSERT INTO [ExampleEntities]
   ([MyMessage]) VALUES (@p0);
```

Вставляет (создает) новую строку в таблицу ExampleEntities

```
SELECT [ExampleEntityId]
FROM [ExampleEntities]
WHERE @@ROWCOUNT = 1 AND
     [ExampleEntityId] = scope_identity();
```

Считывает первичный ключ во вновь созданной строке

Вторая SQL-команда, созданная EF Core, считывает первичный ключ строки, созданной сервером базы данных. Эта команда обеспечивает обновление первичного ключа у исходного экземпляра `ExampleEntity`, чтобы версия сущности в памяти совпадала с версией в базе данных. Чтение первичного ключа важно, потому что вы можете обновить экземпляр сущности позже, а для обновления экземпляра в базе данных потребуется первичный ключ.

> **EF6** При вызове метода `SaveChanges` EF6.x по умолчанию проверяет данные, используя стандартный подход к валидации .NET; он ищет атрибуты проверки данных и, если они присутствуют, вызывает `IValidatableObject.Validate` для классов сущностей. EF Core не включает эту функцию из-за большого количества проверок, выполняемых во внешнем интерфейсе, но при необходимости добавить функцию проверки нетрудно. В главе 4 показано, как это сделать.

3.2.2 Создание книги с отзывом

Теперь посмотрим на создание, включающее в себя связи – в данном случае это добавление новой книги с отзывом. Хотя создание классов сущностей немного сложнее, у этого процесса те же шаги, что и у нашего предыдущего варианта без связей:

- неким образом класс(ы) сущностей добавляе(ю)тся к отслеживаемым сущностям EF Core с состоянием `Added`;
- вызывается метод `SaveChanges`, который проверяет состояние всех отслеживаемых сущностей и выполняет SQL-команду `INSERT` для всех сущностей с состоянием `Added`.

В этом примере используется база данных приложения Book App с таблицами Books и Reviews. На рис. 3.1 показана частичная диаграмма базы данных этих таблиц.

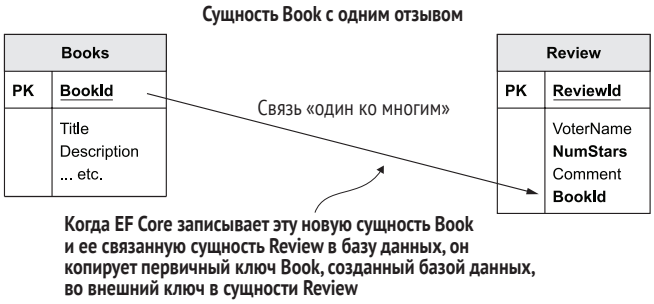


Рис. 3.1 Таблицы Books и Reviews. У строки Review есть внешний ключ, который EF Core заполняет значением первичного ключа из созданной новой строки таблицы Books

В следующем листинге мы создаем новую сущность `Book` и добавляем в свойство коллекции `Reviews` сущность `Review`. Затем вызываем метод `context.Add`, за которым следует метод `SaveChanges`, записывающий обе сущности в базу данных.

Листинг 3.3 При добавлении класса сущности Book также добавляются все связанные классы сущностей

```
var book = new Book
{
    Title = "Test Book",
    PublishedOn = DateTime.Today,
    Reviews = new List<Review>()
    {
        new Review
        {
            NumStars = 5,
            Comment = "Great test book!",
            VoterName = "Mr U Test"
        }
    }
};

context.Add(book);
context.SaveChanges();
```

Создает книгу под названием Test Book

Создает новую коллекцию отзывов

Добавляет один отзыв с содержимым

Использует метод Add, чтобы добавить книгу в свойство DbContext приложения, Books

Вызывает метод SaveChanges из DbContext приложения для обновления базы данных. Он находит новую книгу, в которой есть коллекция, содержащая один новый отзыв, а затем добавляет обе сущности в базу данных

Здесь следует отметить, что мы добавляем только класс сущности `Book`, но связанный класс сущности `Review` также записывается в базу данных. Это происходит из-за того, что EF Core совершает обход по всем связям и находит новый экземпляр `Review`, а поскольку этот экземпляр пока не отслеживается, EF Core знает, что его нужно добавить в базу данных.

Как было показано в простом примере из листинга 3.1, EF Core определяет, что делать со связанными классами сущностей, получая доступ к их значениям `State`. Если это новые связанные экземпляры (еще неизвестные EF Core), то EF Core начнет их отслеживать и задает для их свойства `State` значение `Added`. Во всех остальных случаях EF Core будет подчиняться свойству `State`, привязанному к экземпляру сущности. В листинге 3.3 экземпляр сущности `Review` еще не известен EF Core, а это означает, что значение его свойства `State` – `Detached`, но при вызове метода `Add` задается значение `Added`. Данный экземпляр будет вставлен в базу данных как новая строка.

ЧТО ПРОИСХОДИТ ПОСЛЕ УСПЕШНОГО ЗАВЕРШЕНИЯ РАБОТЫ МЕТОДА SAVECHANGES?

После успешного завершения работы методов `Add` и `SaveChanges` происходит следующее: экземпляры сущностей, которые были вставлены в базу данных, теперь отслеживаются EF Core, и для их состояния устанавливается значение `Unchanged`. Поскольку мы используем реляционную базу данных, а у двух классов сущностей, `Book` и `Review`, есть первичные ключи типа `int`, EF Core по умолчанию ожидает, что база данных создаст первичные ключи с помощью ключевого слова SQL `IDENTITY`. Таким образом, команды SQL, созданные EF Core, считывают первичные ключи в соответствующие первичные ключи в экземплярах классов сущностей, чтобы убедиться, что классы сущностей соответствуют базе данных.

> **ПРИМЕЧАНИЕ** У базы данных Cosmos DB нет аналога `IDENTITY`, поэтому необходимо предоставить уникальный ключ, например GUID (глобальный уникальный идентификатор). Уникальные идентификаторы GUID генерируются тем, что EF Core называет `ValueGenerator` (см. главу 10). GUID также полезны для первичных ключей в реляционных базах данных, когда вам нужен уникальный ключ, который не будет меняться при копировании или дублировании данных в другую базу данных.

Кроме того, EF Core знает о связях по навигационным свойствам в классах сущностей. В листинге 3.3 у свойства `Reviews` сущности `Book` в коллекции есть новый экземпляр сущности `Review`. В рамках процесса `SaveChanges` любой внешний ключ можно будет установить путем копирования первичных ключей во внешние ключи в каждой новой связи. Тогда экземпляр сущности будет соответствовать базе данных. Это полезно, если вы хотите прочитать первичный или внешний ключ, а EF Core сможет обнаружить все последующие изменения, которые вы вносите, если снова вызовете метод `SaveChanges`.

Почему нужно вызывать метод SaveChanges только один раз в конце изменений

В листинге 3.3. видно, что метод SaveChanges вызывается в конце операции создания, и вы увидите тот же шаблон – он вызывается в конце – и в примерах с обновлением и удалением. Фактически даже для сложных изменений в базе данных, где сочетаются операции создания, обновления и удаления, все равно нужно вызывать метод SaveChanges только один раз в конце, потому что EF Core сохранит все ваши изменения (создание, обновление и удаление) и применит их к базе данных вместе, а если база данных отклонит какое-либо из ваших изменений, то будут отклонены все изменения (посредством функции базы данных, называемой *транзакцией*; см. раздел 4.7.2).

Этот паттерн называется *Единицей работы* (*Unit Of Work*). Он означает, что изменения в базе данных нельзя применить наполовину. Например, если вы создали новую сущность Book со ссылкой BookAuthor на Author, которого нет в базе данных, вы не хотите, чтобы экземпляр Book был сохранен. Если сохранить его, то это может нарушить привычное отображение книг. Ведь у каждой книги должен быть хотя бы один автор.

Иногда вы можете подумать, что вам нужно вызвать метод SaveChanges дважды, например когда нужно получить первичный ключ созданного экземпляра сущности для заполнения внешнего ключа у связанного экземпляра сущности, но при работе с EF Core всегда есть выход из этой ситуации. Фактически листинг 3.3 обходит это, создавая новую книгу и новый отзыв одновременно. Прочтите разделы 6.2.1 и 6.2.2, чтобы получить представление о том, как EF Core решает эту задачу.

ПРИМЕР, У КОТОРОГО УЖЕ ЕСТЬ ОДИН ЭКЗЕМПЛЯР В БАЗЕ ДАННЫХ

Другая ситуация, с которой вам, возможно, придется иметь дело, – это создание новой сущности, содержащей навигационное свойство, которое использует еще одну сущность, уже находящуюся в базе данных. Если хотите создать новую сущность Book, у которой есть автор, уже присутствующий в базе данных, необходимо получить отслеживаемый экземпляр сущности Author, который вы хотите добавить в свою новую сущность Book. В следующем листинге приведен один пример. Обратите внимание, что в базе данных уже содержится автор по имени «Mr. A.».

Листинг 3.4 Добавляем книгу с уже существующим автором

```
var foundAuthor = context.Authors
    .SingleOrDefault(author => author.Name == "Mr. A");
if (foundAuthor == null)
    throw new Exception("Author not found");
```

Читает и проверяет, что автор найден

```
var book = new Book
{
    Title = "Test Book",
    PublishedOn = DateTime.Today
};
book.AuthorsLink = new List<BookAuthor>
{
    new BookAuthor
    {
        Book = book,
        Author = foundAuthor
    }
};
```

Создает сущность Book, как и в предыдущем примере

Добавляет связующую запись AuthorBook, но использует автора, который уже есть в базе данных

```
context.Add(book);
context.SaveChanges();
```

Добавляет новую сущность Book в DbContext и свойство Books и вызывает метод SaveChanges

Первые четыре строки загружают сущность `Author` с проверками, чтобы убедиться, что он найден; этот экземпляр класса `Author` отслеживается, поэтому EF Core знает, что он уже находится в базе данных. Мы создаем новую сущность `Book` и добавляем новую связующую сущность `BookAuthor`, но вместо того, чтобы создавать новый экземпляр сущности `Author`, мы используем сущность `Author`, которую прочитали из базы данных. Поскольку EF Core отслеживает экземпляр `Author` и знает, что он находится в базе данных, EF Core не будет пытаться снова добавить его в базу данных при вызове метода `SaveChanges` в конце листинга 3.4.

3.3 Обновление строк базы данных

Обновление строки базы данных осуществляется в три этапа:

1. чтение данных (строки базы данных), возможно, с некоторыми связанными данными;
2. изменение одного или нескольких свойств (столбцы базы данных);
3. запись изменений обратно в базу данных (обновление строки).

В этом разделе мы будем игнорировать все связи и сосредоточимся на этих трех этапах. В следующем разделе вы узнаете, как обновлять связи, добавляя дополнительные команды на каждом этапе.

В листинге 3.5 дата публикации существующей книги изменяется. Используя этот код, можно увидеть стандартный поток обновления.

1. Вы загружаете класс(ы) сущности, который(е) хотите изменить, как отслеживаемую сущность.
2. Вы изменяете свойство/свойства в своем(их) классе(ах) сущности.
3. Вы вызываете метод `SaveChanges` для обновления базы данных.

Листинг 3.5 Обновление даты публикации книги «*Quantum Networking*»

```
var book = context.Books
    .SingleOrDefault(p =>
        p.Title == "Quantum Networking");
if (book == null)
    throw new Exception("Book not found");

book.PublishedOn = new DateTime(2058, 1, 1);
context.SaveChanges();
```

Находит конкретную книгу, которую вы хотите обновить, – в данном случае нашу специальную книгу Quantum Networking

Выбрасывает исключение, если книга не найдена

Изменяет ожидаемую дату публикации на 2058 г. (было 2057 г.)

Вызывает метод SaveChanges, который включает в себя вызов метода DetectChanges. Данный метод обнаруживает, что свойство PublishedOn было изменено

Когда вы вызываете метод `SaveChanges`, он вызывает метод `DetectChanges`, который сравнивает отслеживаемый снимок с экземпляром класса сущности, переданным приложению при первоначальном выполнении запроса. В этом примере EF Core решил, что было изменено только свойство `PublishedOn`, и он создает SQL-запрос для обновления этого свойства.

> **ПРИМЕЧАНИЕ** Использование снимка отслеживания – обычный способ, с помощью которого метод `DetectChanges` находит измененные свойства. Но в главе 11 описана альтернатива снимку отслеживания, например `INotifyPropertyChanging`. Это продвинутая тема, поэтому я использую подход с отслеживаемыми сущностями на протяжении всей первой части книги.

В следующем листинге показаны две SQL-команды, которые EF Core создает для кода из листинга 3.5. Одна команда находит и загружает класс сущности `Book`, а вторая обновляет столбец `PublishedOn`.

Листинг 3.6 SQL-запрос, сгенерированный EF Core для запроса и обновления в листинге 3.5

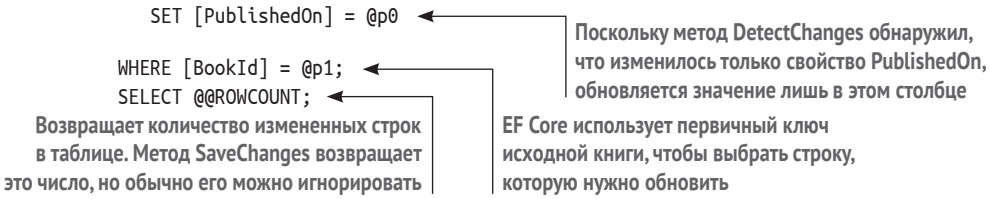
```
SELECT TOP(2)
    [p].[BookId],
    [p].[Description],
    [p].[ImageUrl],
    [p].[Price],
    [p].[PublishedOn],
    [p].[Publisher],
    [p].[Title]
FROM [Books] AS [p]
WHERE [p].[Title] = N'Quantum Networking'

SET NOCOUNT ON;
UPDATE [Books]
```

Читает до двух строк из таблицы Books. Вы запросили один элемент, но этот код гарантирует, что он не сработает, если будет считано две строки

При чтении загружаются перечисленные в SQL-запросе столбцы таблицы

Метод LINQ Where выбирает строки с правильным заголовком

SQL-команда UPDATE – в данном случае для обновления данных таблицы Books



3.3.1 Обработка отключенных обновлений в веб-приложении

Из раздела 3.3 вы уже узнали, что обновление состоит из трех этапов: чтение данных; изменение данных; вызов метода `SaveChanges` для сохранения. Все этапы должны быть выполнены через один и тот же экземпляр DbContext приложения. Проблема состоит в том, что для некоторых приложений, таких как сайты и REST API, использование одного и того же экземпляра DbContext приложения невозможно, потому что в веб-приложениях каждый HTTP-запрос обычно представляет собой новый запрос без каких-либо данных предыдущего запроса. В таких приложениях обновление состоит из двух этапов:

1. Первый этап – это начальное чтение, выполняемое в одном экземпляре DbContext приложения.
2. На втором этапе применяется обновление с использованием нового экземпляра DbContext приложения.

В EF Core такой тип обновления называется *отключенным*, потому что на первом и на втором этапах используются два разных экземпляра DbContext приложения (см. предыдущий список). Отключенное обновление можно обработать несколькими способами. Способ, который нужно использовать, во многом зависит от вашего приложения. Вот два основных варианта обработки отключенных обновлений:

- *вы отправляете только те данные, которые вам нужно обновить, из первого этапа*. Если бы вы обновляли дату публикации книги, то должны были бы отправить обратно только свойства `BookId` и `PublishedOn`. На втором этапе вы используете первичный ключ для загрузки исходной сущности с отслеживанием и обновляете необходимые свойства. В данном примере первичный ключ – это `BookId`, а свойство для обновления – это `PublishedOn` сущности `Book` (см. рис. 3.2). Когда вы вызываете метод `SaveChanges`, EF Core может определить, какие свойства вы изменили, и обновить только эти столбцы в базе данных;
- *вы отправляете все данные, необходимые для создания копии объекта класса сущности, из первого этапа*. На втором этапе создаете объект класса сущности и, возможно, связи, используя данные из первого этапа, и сообщаете EF Core обновить всю сущность (см. рис. 3.3). Когда вы вызываете метод `SaveChanges`, EF Core будет знать, что он должен обновить все столбцы в строке (строках) таблицы, которые соответствуют объектам, полученным на первом этапе.

ПРИМЕЧАНИЕ Еще один способ частичного обновления сущности, описанный в варианте 1, – создать новый экземпляр сущности и управлять состоянием каждого свойства экземпляра. Этот вариант рассматривается в главе 11, когда мы будем более подробно изучать, как изменить состояние сущности.

Сколько текста! Теперь я приведу пример каждого подхода при работе с отключенными обновлениями.

ОТКЛЮЧЕННОЕ ОБНОВЛЕНИЕ С ПОВТОРНОЙ ЗАГРУЗКОЙ

На рис. 3.2 показан пример отключенного обновления в веб-приложении. В этом случае вы предоставляете возможность пользователю с правами администратора обновить дату публикации книги. На рисунке показано, что обратно отправляются только данные `BookId` и `PublishDate` из первого этапа.

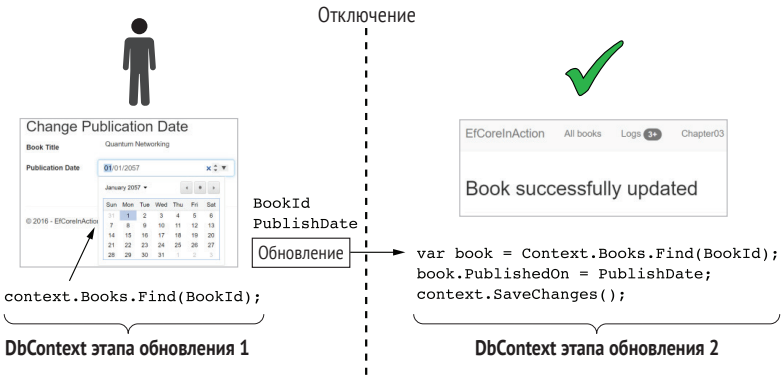


Рис. 3.2 Два этапа выполнения отключенного обновления на сайте с использованием EF Core. Пунктирная линия посередине представляет собой точку, в которой данные, хранящиеся в приложении на первом этапе, теряются, а второй этап начинается, не зная, что было сделано на первом этапе. Когда пользователь нажимает кнопку «Обновить», возвращаются только сведения `BookId` и `PublishDate`

Для веб-приложений обычным подходом для обновления данных через EF Core является возврат на веб-сервер ограниченного объема данных, достаточного для обновления. Такой подход делает запрос быстрее, но главная причина – это безопасность. Например, вы бы не хотели, чтобы при обновлении даты публикации книги изменялась и ее цена, поскольку эту информацию могут изменить хакеры.

Есть несколько способов контролировать, какие данные возвращаются или принимаются веб-сервером. Например, в ASP.NET Core есть атрибут `BindNever`, позволяющий определять именованные свойства, которые не будут возвращены во второй этап. Но более общий подход, который предпочитаю я, заключается в использовании специ-

ального класса, содержащего только те свойства, которые должны быть отправлены или получены. Этот класс называется Data Transfer Object (DTO) или ViewModel. По своей природе он похож на DTO, используемый в выборочном запросе в главе 2, но здесь он применяется не только в запросе, но и для получения нужных вам данных, от пользователя через браузер. Для нашего примера, где мы обновляем дату публикации, понадобятся три части. Первая часть, DTO для отправки и получения данных пользователю и от него, показана здесь.

Листинг 3.7 Класс `ChangePubDateDto` для отправки данных пользователю и получения их от него

```
public class ChangePubDateDto
{
    public int BookId { get; set; }

    public string Title { get; set; }

    [DataType(DataType.Date)]
    public DateTime PublishedOn { get; set; }
}
```

Содержит первичный ключ строки, которую вы хотите обновить, что позволяет быстро найти нужную строку

Вы отправляете заголовок, чтобы показать пользователю, что он может быть уверен, что изменяет правильную книгу

Свойство, которое вы хотите изменить. Вы отправляете текущую дату публикации и получаете обратно измененную дату

Самый быстрый способ прочитать класс сущности с помощью его первичного ключа (ключей)

Если вы хотите обновить конкретную сущность и вам нужно получить ее, используя первичный ключ, то есть несколько вариантов. Раньше я использовал метод `Find`, но, покопавшись, теперь я рекомендую метод `SingleOrDefault`, потому что он быстрее. Однако нужно отметить два полезных момента, касающихся метода `Find`:

- метод `Find` проверяет DbContext текущего приложения, чтобы узнать, был ли уже загружен требуемый экземпляр сущности. Это позволяет избежать запроса к базе данных. Но если в DbContext приложения сущности нет, загрузка будет идти медленнее из-за этой дополнительной проверки;
- имя метода `Find` проще и быстрее набирать на клавиатуре, потому что оно короче, чем `SingleOrDefault`. Сравните, например: `context.Find<Book>(key)` и `context.SingleOrDefault(p => p.Bookid == key)`.

Преимущество использования метода `SingleOrDefault` состоит в том, что его можно добавить в конец запроса с такими методами, как `Include`, чего нельзя сделать с помощью `Find`.

Во-вторых, нам нужен метод получения исходных данных для этапа 1. В-третьих, нам нужен метод, чтобы получить данные обратно из браузера, а затем повторно загрузить и обновить книгу. В этом лис-

тинге показан класс `ChangePubDateService`, содержащий два метода для обработки этих двух этапов.

Листинг 3.8 Класс `ChangePubDateService` для обработки отключенного обновления

```
public class ChangePubDateService : IChangePubDateService
{
    private readonly EfCoreContext _context;

    public ChangePubDateService(EfCoreContext context)
    {
        _context = context;
    }
public ChangePubDateDto GetOriginal(int id)
{
    return _context.Books
        .Select(p => new ChangePubDateDto
        {
            BookId = p.BookId,
            Title = p.Title,
            PublishedOn = p.PublishedOn
        })
        .Single(k => k.BookId == id);
}

public Book UpdateBook(ChangePubDateDto dto)
{
    var book = _context.Books.SingleOrDefault(
        x => x.BookId == dto.BookId);
    if (book == null)
        throw new ArgumentException(
            "Book not found");
    book.PublishedOn = dto.PublishedOn;
    _context.SaveChanges();
    return book;
 }
}
```

Данный интерфейс используется для регистрации класса в контейнере внедрения зависимостей. Внедрение зависимостей используется в главе 5 при создании приложения Book App с ASP.NET Core

DbContext приложения передается через конструктор класса – обычный способ создания классов, которые вы будете использовать в качестве сервиса в ASP.NET Core

Метод отвечает за первый этап обновления. Он получает данные из выбранной книги для отображения пользователю

Формирует запрос для получения только трех свойств из строки

Использует первичный ключ для получения определенной строки из базы данных

Метод отвечает за второй этап обновления. Выполняет частичное обновление выбранной книги

Загружает книгу. Я использую SingleOrDefault, потому что он немного быстрее метода Find

Обновляет только свойство PublishedOn

Метод SaveChanges вызывает метод DetectChanges для поиска изменений, а после применяет только их к базе данных

Возвращает обновленную книгу

Я обрабатываю случай, когда книга не была найдена. Генерируется исключение с описанием ошибки

Преимущества этого подхода с повторной загрузкой и последующим обновлением заключаются в том, что он безопаснее (в нашем примере отправка и получение стоимости книги по протоколу HTTP позволили бы подменить ее) и быстрее благодаря меньшему количеству данных. Обратная сторона состоит в том, что вам необходимо писать дополнительный код для копирования определенных свойств, которые вы хотите обновить. В главе 6 приводится несколько приемов для автоматизации этого процесса.

ПРИМЕЧАНИЕ Вы можете изучить этот код и самостоятельно попробовать обновить дату публикации в приложении Book App. Если вы скачаете код из git репозитория и запустите его локально, то увидите кнопку **Admin** для каждой книги. Эта кнопка содержит ссылку **Change Pub Date** (Изменить дату публикации), которая проведет вас через весь процесс. Кроме того, можно ознакомиться с SQL-командами, которые EF Core использует для выполнения обновления, перейдя в пункт меню «Журналы».

ОТКЛЮЧЕННОЕ ОБНОВЛЕНИЕ. ОТПРАВКА ВСЕХ ДАННЫХ

В некоторых случаях все данные могут быть отправлены обратно, поэтому нет необходимости повторно загружать исходные данные. Это может происходить для простых классов сущностей, в некоторых REST API или при обмене данными между процессами (process-to-process communication). Многое зависит от того, насколько соответствует формат данных обмена через API формату базы данных, а также от уровня вашего доверия к другой системе.

На рис. 3.3 показан пример REST API, где внешняя система сначала запрашивает у системы книги с определенным названием. На этапе обновления внешняя система отправляет обратно обновленную информацию об авторе полученной книги.

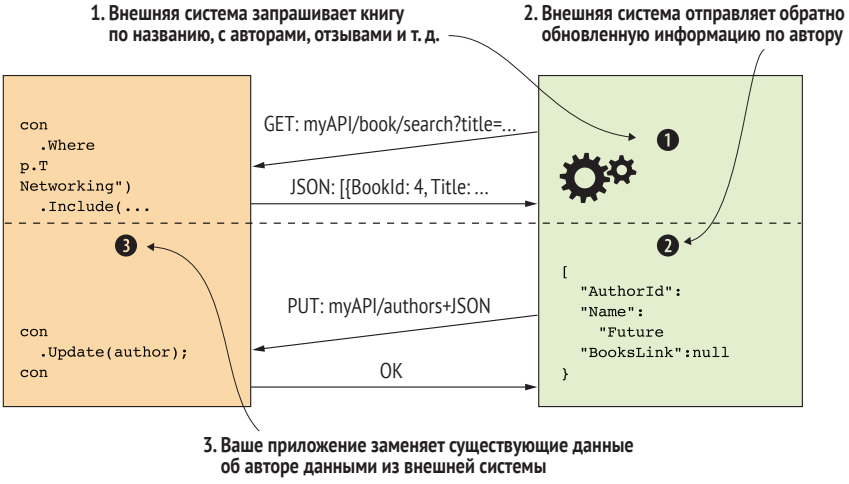


Рис. 3.3 Пример отключенного обновления, где вся информация в выбранных строках базы данных полностью заменяется новыми данными. В отличие от предыдущего примера, этот процесс не требует повторной загрузки данных перед обновлением

В листинге 3.9 имитируются запросы REST API. На первом этапе вы получаете экземпляр класса сущности `Author`, который хотите обновить, а затем сериализуете его в строку формата JSON. (На рис. 3.3, на

этапе 2 показан пример JSON-строки.) На втором этапе вы декодируете JSON-строку в экземпляр класса сущности и используете команду EF Core `Update`, заменяющую всю информацию в строке, первичный ключ которой совпадает со значением `AuthorId`.

Листинг 3.9 Моделирование запроса внешней системы на обновление/замену

```
string json;
using (var context = new EfCoreContext(options))
{
    var author = context.Books
        .Where(p => p.Title == "Quantum Networking")
        .Select(p => p.AuthorsLink.First().Author)
        .Single();
    author.Name = "Future Person 2";
    json = JsonConvert.SerializeObject(author);
}
using (var context = new EfCoreContext(options))
{
    var author = JsonConvert
        .DeserializeObject<Author>(json);

    context.Authors.Update(author);
    context.SaveChanges();
}
```

Имитирует внешнюю систему, возвращающую измененный класс сущности Author в виде JSON-строки

Имитирует получение JSON-строки от внешней системы и ее декодирование в экземпляр класса Author

Команда обновления, которая заменяет все данные строки, первичный ключ которой совпадает со значением AuthorId

Предоставляет ссылку на промежуточную таблицу, необходимую для связи «многие ко многим», в которой есть ссылки на авторов этой книги

Мы вызываем метод `Update` с экземпляром сущности `Author` в качестве параметра. Метод помечает все свойства экземпляра как измененные. Когда мы вызываем метод `SaveChanges`, он обновляет все столбцы в строке, у которой первичный ключ как в классе сущности.

> **EF6** Метод `Update` впервые появился в EF Core. В EF6.x нужно управлять состоянием объекта напрямую, например с помощью команды `DbContext.Entry(object).State = EntityState.Modified`. Незаметные изменения в способе настройки состояния объекта, используемом EF Core, описаны в главе 11.

Плюс такого подхода состоит в том, что обновление базы данных происходит быстрее, потому что не нужно дополнительно читать исходные данные. Кроме того, вам не нужно писать код для обновления определенных свойств объекта, в отличие от предыдущего подхода, где вам пришлось бы это делать.

Недостаток здесь состоит в том, что нужно передавать больше данных, и если API плохо спроектирован, возможно, будет непросто со-

гласовать данные, которые вы получаете, с данными, уже находящимися в базе. Кроме того, вы доверяете внешней системе правильно запоминать все данные, особенно первичные ключи вашей системы.

ПРИМЕЧАНИЕ В листинге 3.9 описывается только один класс без связи, но во многих REST API и при обмене данными между процессами можно отправлять много связанных данных. В этом примере API может ожидать, что вся книга со всеми связями будет отправлена обратно только для обновления имени автора. Это сложный процесс, поэтому я расскажу о нем в главе 11, где показано, как управлять состоянием каждого свойства, и рассказывается о методе EF Core `TrackGraph`, который помогает обрабатывать частичные обновления классов со связями.

3.4 Обработка связей в обновлениях

Теперь, когда мы определили три основных шага по обновлению базы данных, пора взглянуть на обновление связей между классами сущностей, например добавив новый отзыв на книгу. Обновление связей добавляет еще один уровень сложности в код, особенно в отключенном состоянии, поэтому я вынес это в отдельный раздел.

В этом разделе описаны обновления для трех типов связей, которые используются в EF Core, и приводятся примеры как подключенных, так и отключенных обновлений. Во всех случаях мы будем использовать класс сущности `Book`, у которого три связи. В следующем листинге показан класс сущности `Book`, но с акцентом на связи. (Я удалил некоторые свойства без связей для упрощения.)

Листинг 3.10 Класс сущности Book, демонстрирующий связи для обновления

```
public class Book
{
    public int BookId { get; set; }
    //… Остальные свойства без связей были удалены для ясности;
    //-----------------------------------------------
    //Связи;

    public PriceOffer Promotion { get; set; }
    public ICollection<Review> Reviews { get; set; }
    public ICollection<Tag> Tags { get; set; }
    public ICollection<BookAuthor>
        AuthorsLink { get; set; }
}
```

Класс Book содержит основную информацию о книге

Ссылка на необязательное PriceOffer

Количество отзывов на книгу может быть от 0 до множества

EF Core 5 автоматически создает связь «многие ко многим» между классами сущностей Tag и Book

Предоставляет ссылку на таблицу, хранящую связь «многие ко многим», которая содержит ссылки на авторов этой книги

3.4.1 Основные и зависимые связи

Термины *основной* и *зависимый* используются в EF для определения частей связи:

- *основная сущность* – содержит первичный ключ, на который ссылается зависимая сущность через внешний ключ;
- *зависимая сущность* – содержит внешний ключ, который ссылается на первичный ключ основной сущности.

В приложении Book App класс сущности `Book` является основным. Классы сущностей `PriceOffer`, `Review` и `BookAuthor` – зависимыми. Я считаю термины *основной* и *зависимый* полезными, потому что они определяют, кто здесь главный: основная сущность. Я использую эти термины в книге там, где это применимо.

> **ПРИМЕЧАНИЕ** Класс сущности может быть как основной, так и зависимой сущностью одновременно. В иерархической связи, скажем, библиотеки с книгами, на которые есть отзывы, книга будет зависимой связью класса сущности библиотеки.

МОЖЕТ ЛИ ЗАВИСИМАЯ ЧАСТЬ СВЯЗИ СУЩЕСТВОВАТЬ БЕЗ ОСНОВНОЙ ЧАСТИ?

Другой аспект зависимой связи – может ли она существовать сама по себе. Если основная связь удалена, есть ли необходимость для дальнейшего существования зависимой связи? Во многих случаях зависимая часть связи не имеет смысла без основной связи. Например, рецензия на книгу не имеет смысла, если эта книга удалена из базы данных.

В некоторых случаях зависимая связь должна существовать, даже если основная часть была удалена. Предположим, вы хотите вести журнал всех изменений, которые происходят с книгой. Если вы удалите ее, то вряд ли захотите, чтобы этот набор записей журнала тоже был удален.

В базе данных эта задача решается допустимостью значения `null` для внешнего ключа. Если внешний ключ в зависимой связи не допускает значения `null`, то эта связь не может существовать без основной части. В базе данных приложения Book App сущности `PriceOffer`, `Review` и `BookAuthor` зависят от основной части, сущности `Book`, поэтому их внешние ключи имеют тип `int`. Если книга или ссылка на книгу была удалена, зависимые сущности также будут удалены.

Но если вы определяете класс для журналирования – назовем его `BookLog`, – вам нужно, чтобы этот класс остался, даже если книга была удалена. Для этого необходимо сделать так, чтобы внешний ключ `BookId` имел тип `Nullable<int>`. Затем, если вы удалите книгу, с которой связана сущность `BookLog`, можно сделать так, что внешний ключ `BookLog`, `BookId`, примет значение `null`.

ПРИМЕЧАНИЕ В предыдущем примере с BookLog, если вы удалите сущность Book, с которой связана сущность BookLog, действием по умолчанию можно будет задать для внешнего ключа BookLog значение null, потому что EF Core по умолчанию использует параметр ClientSetNull для свойства необязательных связей OnDelete. В разделе 8.8.1 эта тема рассматривается более подробно.

Я упоминаю об этой ситуации сейчас, потому что, когда мы обновляем связи, в некоторых случаях зависимая связь удаляется из основной. Приведу пример замены всех зависимых связей новыми. Что происходит со старыми связями, которые мы удаляем, зависит от того, допускает ли внешний ключ значение null: если нет, то зависимые связи удаляются, иначе для него устанавливается значение null. Подробнее об этой теме и о том, как EF Core обрабатывает удаление, я расскажу в разделе 3.5.

3.4.2 Обновление связей «один к одному»: добавляем PriceOffer в книгу

В нашей базе данных приложения Book App в классе сущности Book есть необязательное свойство Promotion с зависимой связью для класса сущности PriceOffer. В этом подразделе рассказывается, как добавить класс PriceOffer к существующей книге. В следующем листинге показана структура класса сущности PriceOffer, который связан с таблицей Books с помощью внешнего ключа BookId.

Листинг 3.11 Класс сущности PriceOffer, связанный с таблицей Books с помощью внешнего ключа

```
public class PriceOffer
{
    public int PriceOfferId { get; set; }
    public decimal NewPrice { get; set; }
    public string PromotionalText { get; set; }

    //------------------------------------------------
    //Связи;

    public int BookId { get; set; }
}
```

PriceOffer, если присутствует, предназначен для замены обычной цены на акционную

Внешний ключ для книги, к которой он должен быть применен

ОБНОВЛЕНИЕ ПРИ ПОДКЛЮЧЕННОМ СОСТОЯНИИ

Обновление при подключенном состоянии предполагает, что вы используете один и тот же контекст данных (context) для чтения и обновления. В листинге 3.12 показан пример кода, в котором есть три этапа:

1. загрузка сущности Book с зависимой сущностью PriceOffer;
2. установка связи с новой сущностью PriceOffer, которую вы хотите связать с этой книгой;
3. вызов метода SaveChanges, чтобы обновить базу данных.

Листинг 3.12 Добавляем новую акционную цену для существующей книги, у которой ее нет

Находит книгу. В этом примере для книги нет рекламной акции, но все работало бы так же, если бы она была

Хотя метод Include не требуется, потому что вы выполняете загрузку книги без рекламной акции, но его использование является хорошей практикой, так как следует загружать любые связи, если вы собираетесь их изменить

```
var book = context.Books
    .Include(p => p.Promotion)
    .First(p => p.Promotion == null);

book.Promotion = new PriceOffer
{
    NewPrice = book.Price / 2,
    PromotionalText = "Half price today!"
};
context.SaveChanges();
```

Добавляет новый экземпляр PriceOffer для книги

Метод SaveChanges вызывает метод DetectChanges, который обнаруживает, что свойство Promotion было изменено, и добавляет эту сущность в таблицу PriceOffers

Как видите, обновление связи похоже на обычное обновление, которое вы выполнили, чтобы изменить дату публикации книги. В этом случае EF Core приходится выполнять дополнительную работу из-за связи. В таблице PriceOffers создается новая строка, которую видно во фрагменте SQL-команды, которую EF Core создает для кода из листинга 3.12:

```
INSERT INTO [PriceOffers]
    ([BookId], [NewPrice], [PromotionalText])
     VALUES (@p0, @p1, @p2);
```

Теперь что произойдет, если для книги действует рекламная акция (т. е. свойство Promotion из класса сущности Book не равно null)? Вот почему метод Include(p => p.Promotion) в запросе, который загрузил класс сущности Book, так важен. Благодаря методу Include EF Core будет знать про существующий у книги экземпляр PriceOffer и удалит его перед добавлением новой версии.

Чтобы было ясно, в этом случае нужно использовать одну из форм загрузки связей – *немедленную* (*eager*), *явную* (*explicit*), *выборочную* (*select*) или *отложенную* (*lazy loading*), чтобы EF Core знала об этом перед обновлением. Если вы этого не сделаете, а связь существует, то EF Core сгенерирует исключение для дублирующегося значения внешнего ключа BookId, для которого EF Core назначила уникальный индекс в базе данных, а другая строка в таблице PriceOffers будет иметь такое же значение.

ОБНОВЛЕНИЕ ПРИ ОТКЛЮЧЕННОМ СОСТОЯНИИ

В отключенном состоянии информация, позволяющая определить, какую книгу нужно обновить и что поместить в класс сущности `PriceOffer`, будет передаваться из этапа 1 в этап 2. Такой же сценарий произошел при обновлении даты публикации книги (рис. 3.2), когда были возвращены только значения `BookId` и `PublishedOn`.

В случае добавления рекламной акции для книги необходимо передать `BookId`, который однозначно определяет нужную книгу, а также значения `NewPrice` и `PromotionalText`, составляющие класс сущности `PriceOffer`. В следующем листинге показан класс `ChangePriceOfferService`, который содержит два метода для получения данных, отображаемых пользователю, и обновления рекламной акции в классе сущности `Book`, когда пользователь отправляет запрос.

Листинг 3.13 Класс `ChangePriceOfferService` с методом для обработки каждого этапа

```
public class ChangePriceOfferService : IChangePriceOfferService
{
    private readonly EfCoreContext _context;

    public Book OrgBook { get; private set; }

    public ChangePriceOfferService(EfCoreContext context)
    {
        _context = context;
    }

    public PriceOffer GetOriginal(int id)
    {
        OrgBook = _context.Books
            .Include(r => r.Promotion)
            .Single(k => k.BookId == id);

        return OrgBook?.Promotion
            ?? new PriceOffer
            {
                BookId = id,
                NewPrice = OrgBook.Price
            };
    }

    public Book AddUpdatePriceOffer(PriceOffer promotion)
    {
        var book = _context.Books
            .Include(r => r.Promotion)
            .Single(k => k.BookId
                == promotion.BookId);

        if (book.Promotion == null)
```

Получает и отправляет экземпляр класса PriceOffer пользователю для обновления

Загружает книгу с текущей рекламной акцией

Обрабатывает вторую часть обновления, выполняя выборочное добавление/обновление свойства Promotion выбранной книги

Вы либо возвращаете существующую рекламную акцию для редактирования, либо создаете новую. Важным моментом является установка значения ключа BookId, так как вам нужно передать его для второго этапа

Загружает книгу с текущей рекламной акцией, что важно, потому что в противном случае новый PriceOffer будет конфликтовать и выдаст ошибку

Проверяет, должен ли быть создан новый PriceOffer или обновлен существующий

```
        {
            book.Promotion = promotion;
        }
        else
        {
            book.Promotion.NewPrice
                = promotion.NewPrice;
            book.Promotion.PromotionalText
                = promotion.PromotionalText;
        }
        _context.SaveChanges();
        return book;
    }
}
```

Необходимо добавить новый PriceOffer, чтобы связать рекламную акцию и книгу. EF Core увидит это и добавит новую строку в таблицу PriceOffer

Нужно выполнить обновление, поэтому копируются только те части, которые вы хотите изменить. EF Core увидит это обновление и создаст код для обновления только этих двух столбцов

Метод SaveChanges вызывает метод DetectChanges, который видит, что изменяется – добавляется новый PriceOffer или обновляется существующий

Возвращает обновленную книгу

Этот код обновляет существующий `PriceOffer` либо добавляет новый, если его нет. Когда вы вызываете метод `SaveChanges` с помощью метода EF Core `DetectChanges`, он может определить, какой тип обновления требуется, и создать правильный SQL для обновления базы данных. Это отличается от версии, показанной в листинге 3.12, где мы заменили `PriceOffer` полностью на новую версию. Обе версии работают, но если вы ведете журнал, кто последним создал или обновил сущность (см. раздел 11.4.3), то обновление имеющейся сущности дает немного больше информации о том, что изменилось.

АЛЬТЕРНАТИВНЫЙ СПОСОБ ОБНОВЛЕНИЯ СВЯЗИ: НЕПОСРЕДСТВЕННОЕ СОЗДАНИЕ НОВОЙ СТРОКИ

Мы рассмотрели обновление как изменение связи в классе сущности `Book`, но здесь также можно использовать создание или удаление строки в таблице PriceOffers. В листинге 3.14 сначала находим первую книгу в базе данных, без связи с `Promotion`, и затем добавляем в эту книгу новую сущность `PriceOffer`.

Листинг 3.14 Создание строки `PriceOffer` для существующей книги

```
var book = context.Books
    .First(p => p.Promotion == null);

context.Add( new PriceOffer
{
    BookId = book.BookId,
    NewPrice = book.Price / 2,
    PromotionalText = "Half price today!"
});
context.SaveChanges();
```

Вы находите книгу без PriceOffer, для которой хотите его добавить

Добавляет новый PriceOffer в таблицу PriceOffers

Создаете PriceOffer. Вы должны установить значение BookId (который EF Core заполнил ранее)

SaveChanges добавляет PriceOffer в таблицу PriceOffers

Обратите внимание, что ранее вам не нужно было задавать свойство `BookId` в классе сущности `PriceOffer`, потому что EF Core делал это за вас. Но когда вы создаете связь таким образом, нужно явно

установить внешний ключ. После выполнения кода, если вы загрузите класс сущности `Book` с его зависимой сущностью `Promotion`, то обнаружите, что у `Book` появилась связь `Promotion`.

> **ПРИМЕЧАНИЕ** Класс сущности `PriceOffer` не имеет свойства для связи с классом `Book` (например, `public Book BookLink {get; set;}`). Если бы оно было, то можно было бы установить как значение `BookLink` экземпляр сущности `Book`, вместо того чтобы устанавливать внешний ключ. Установка внешнего ключа (ключей) либо установка связи с основной сущностью сообщит EF Core о необходимости установить связь.

Преимущество создания зависимого класса сущности состоит в том, что это избавляет вас от необходимости заново загружать основной класс сущности (в данном случае `Book`) в отключенном состоянии. Обратная сторона состоит в том, что EF Core не помогает вам со связями. В этом случае, если у книги было существующее ценовое предложение, а вы добавили еще одно, метод `SaveChanges` завершится с ошибкой, потому что у вас будет две строки `PriceOffer` с одинаковым внешним ключом.

Когда EF Core не может помочь вам со связями, нужно осторожно использовать подход с созданием и удалением. Иногда он может упростить работу со сложными связями, поэтому стоит помнить о нем, но я предпочитаю обновлять связь через основную сущность в большинстве случаев связи «один к одному».

> **ПРИМЕЧАНИЕ** Позже, в разделе 3.4.5, вы узнаете еще один способ обновления связей путем изменения внешних ключей.

3.4.3 Обновление связей «один ко многим»: добавляем отзыв в книгу

Мы узнали об основных этапах обновления связей, рассмотрев связь «один к одному». Я рассмотрю оставшиеся связи более бегло, так как вы уже видели базовый шаблон, а также отмечу некоторые различия, касающиеся аспекта *многие*.

Связь «один ко многим» в базе данных приложения Book App представлена отзывами на книгу; пользователь сайта может добавить отзыв к книге. Их количество может быть любым, от нуля до множества. В этом листинге показан класс зависимой сущности, `Review`, который связан с таблицей Books через внешний ключ `BookId`.

Листинг 3.15 Класс `Review`, где показан внешний ключ для связи с таблицей Books

```
public class Review ◄──────── Хранит отзывы покупателей с рейтингами
{
```

```
    public int ReviewId { get; set; }
    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }

    //----------------------------------------
    //Связи;

    public int BookId { get; set; }
}
```

Внешний ключ содержит ключ книги, которой принадлежит этот отзыв

ОБНОВЛЕНИЕ ПРИ ПОДКЛЮЧЕННОМ СОСТОЯНИИ

В листинге 3.16 к книге добавляется новый отзыв. Этот код следует тому же шаблону, что и подключенное обновление «один к одному»: загрузка класса сущности `Book` и связи `Reviews` через метод `Include`. Но в данном случае мы добавляем сущность `Review` в коллекцию `Reviews`. Поскольку мы использовали метод `Include`, свойство `Reviews` будет пустой коллекцией, если отзывов, связанных с этой книгой, нет. В этом примере база данных уже содержит несколько сущностей `Book`, и я возьму первую.

Листинг 3.16 Добавление рецензии к книге в подключенном состоянии

```
var book = context.Books
    .Include(p => p.Reviews)
    .First();

book.Reviews.Add(new Review
{
    VoterName = "Unit Test",
    NumStars = 5,
    Comment = "Great book!"
});
context.SaveChanges();
```

Находит первую книгу и загружает все отзывы, которые могут быть у нее

Добавляет новый отзыв на эту книгу

Метод SaveChanges вызывает метод DetectChanges, который обнаруживает, что свойство Reviews было изменено, и находит новый отзыв, который добавляет в таблицу Review

Как и в примере с `PriceOffer`, внешний ключ (свойство `BookId`) не заполняется в `Review`, потому что EF Core знает, что `Review` добавляется в класс сущности `Book`, и настраивает внешний ключ, используя правильное значение.

ИЗМЕНЕНИЕ/ЗАМЕНА ВСЕХ СВЯЗЕЙ «ОДИН КО МНОГИМ»

Прежде чем перейти к обновлению при отключенном состоянии, я хочу рассмотреть случай, когда нужно изменить или заменить всю коллекцию, вместо того чтобы добавлять в нее что-либо, как мы это сделали с отзывом.

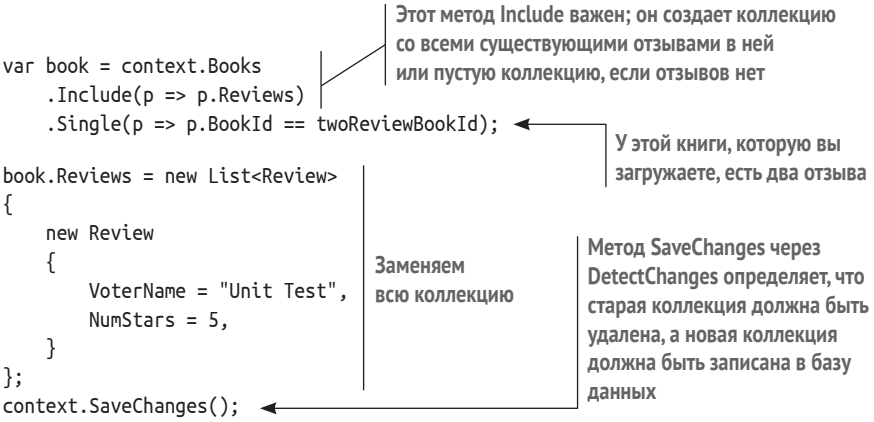
Если в книгах есть категории (скажем, *Проектирование программного обеспечения*, *Языки программирования* и т. д.), то можно разрешить пользователю с правами администратора изменять категории.

Один из способов реализовать это изменение – отображать текущие категории в списке со множественным выбором, разрешить этому пользователю изменять их, а затем заменить *все* категории новыми выбранными.

EF Core упрощает замену всей коллекции. Если вы назначаете новую коллекцию связи «один ко многим», которая была загружена с отслеживанием (например, с помощью метода `Include`), EF Core заменит существующую коллекцию новой. Если элементы в коллекции могут быть связаны только с основным классом (у зависимого класса есть внешний ключ, не допускающий значения null), по умолчанию EF Core удалит элементы, которые были в коллекции.

Далее приводится пример замены всей коллекции существующих отзывов на книгу на новую коллекцию. В результате исходные отзывы удаляются и заменяются одним новым.

Листинг 3.17 Замена всей коллекции отзывов на другую



Поскольку в этом примере мы используем тестовые данные, мы знаем, что книга с первичным ключом `twoReviewBookId` содержит два отзыва и что эта книга – единственная, на которую есть отзывы; следовательно, во всей базе данных только два отзыва. После вызова метода `SaveChanges` у книги только один отзыв, а два старых были удалены, поэтому теперь в базе есть только один отзыв.

Удалить одну строку так же просто, как удалить сущность из списка. EF Core увидит изменение и удалит строку, связанную с этой сущностью. Точно так же, если вы добавите новый отзыв в коллекцию `Reviews`, EF Core увидит это изменение и добавит новый отзыв в базу данных.

Для этих изменений важна загрузка существующей коллекции: если не загрузить их, EF Core не сможет удалить, обновить или заменить их. Старые версии будут по-прежнему находиться в базе данных после обновления, потому что EF Core не знал о них во время обновления. Вы не заменили два существующих отзыва на один. Фактиче-

ски теперь у вас есть три отзыва – два из них изначально были в базе данных, и новый – это не то, что вам было нужно.

ОБНОВЛЕНИЕ ПРИ ОТКЛЮЧЕННОМ СОСТОЯНИИ

В отключенном состоянии вы создаете пустой класс сущности `Review`, но заполняете его внешний ключ, `BookId`, идентификатором книги, для которой пользователь хочет оставить отзыв. Затем пользователь ставит оценку, и вы добавляете этот отзыв к книге, о которой шла речь. В следующем листинге показан класс `AddReviewService`, в котором есть методы для настройки и обновления книги, чтобы добавить новый отзыв от пользователя.

Листинг 3.18 Добавляем новый отзыв к книге в приложении Book App

```
public class AddReviewService
{
    private readonly EfCoreContext _context;

    public string BookTitle { get; private set; }

    public AddReviewService(EfCoreContext context)
    {
        _context = context;
    }

    public Review GetBlankReview(int id)
    {
        BookTitle = _context.Books
            .Where(p => p.BookId == id)
            .Select(p => p.Title)
            .Single();
        return new Review
        {
            BookId = id
        };
    }

    public Book AddReviewToBook(Review review)
    {
        var book = _context.Books
            .Include(r => r.Reviews)
            .Single(k => k.BookId
                == review.BookId);
        book.Reviews.Add(review);
        _context.SaveChanges();
        return book;
    }
}
```

Формирует отзыв для заполнения пользователем

Читаем название книги, которое будет показано пользователю, когда он будет писать отзыв

Создает отзыв с заполненным внешним ключом BookId

Добавляет для книги новый отзыв

Загружает нужную книгу, используя значение внешнего ключа отзыва, и включает все существующие отзывы (или пустую коллекцию, если отзывов еще нет)

Добавляет новый отзыв в коллекцию Reviews

Возвращает обновленную книгу

Метод SaveChanges использует метод DetectChanges, который видит, что свойство Review изменилось, и создает новую строку в таблице Review

Первая часть этого листинга более простая по сравнению с предыдущими примерами с отключенным состоянием, так как вы добавляете новый отзыв, вам не нужно загружать существующие данные. Но в целом здесь используется тот же подход, который использовал класс `ChangePriceOfferService`.

АЛЬТЕРНАТИВНЫЙ СПОСОБ ОБНОВЛЕНИЯ СВЯЗИ: СОЗДАНИЕ НОВОЙ СТРОКИ НАПРЯМУЮ

Как и в случае с `PriceOffer`, можно добавить связь «один ко многим» непосредственно в базу данных. Но опять же, вы берете на себя роль управления связью. Если, например, вы хотите заменить всю коллекцию отзывов, то перед добавлением новой коллекции придется удалить все строки, которыми отзывы связаны с рассматриваемой книгой.

Добавление строки непосредственно в базу данных дает некоторые преимущества, потому что загрузка всех связей «один ко многим» может обернуться большим объемом данных, если у вас большое количество записей и/или они большие по объему. Поэтому помните об этом подходе, если у вас проблемы с производительностью.

> **ПРИМЕЧАНИЕ** Мои эксперименты показывают, что если не загружать связанные данные, а затем присвоить новую коллекцию для свойства класса, которое реализует связь «один ко многим», то это эквивалентно добавлению новой строки напрямую. Но я не рекомендую делать этого, потому что это не привычный паттерн обновления; через некоторое время кто-то другой (или даже вы) может вернуться к этому участку кода и неправильно истолковать логику выполнения.

3.4.4 Обновление связи «многие ко многим»

В EF Core мы говорим о связи «многие ко многим», но реляционная база данных не поддерживает ее реализацию напрямую. Вместо этого мы имеем дело с двумя связями «один ко многим», как показано на рис. 3.4.

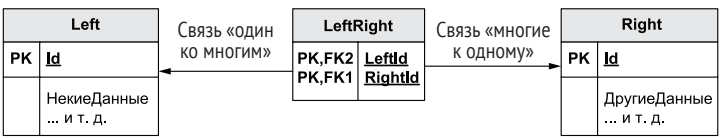


Рис. 3.4 Связь «многие ко многим» в базе данных создается связующей таблицей, содержащей первичные ключи двух таблиц, которым требуется связь «многие ко многим»

В EF Core есть два способа создать связь «многие ко многим» между двумя классами сущностей:

- вы указываете ссылку на связующую таблицу в каждой сущности, т. е. у вас появляется свойство `ICollection<LeftRight>` в классе сущности `Left`. Вам также нужно создать класс сущности, который будет описывать связующую таблицу (как, например, `LeftRight` на рис. 3.4). Этот класс позволяет добавлять дополнительные данные в связующую таблицу, чтобы вы могли сортировать или фильтровать связи «многие ко многим»;
- вы напрямую связываете два класса сущности, для которых нужна связь «многие ко многим», т. е. у вас есть свойство `ICollection<Right>` в классе сущности `Left`. В данном случае писать код намного проще, потому что EF Core занимается созданием связующей таблицы, но тогда вы не сможете получить доступ к этой таблице в методе `Include` для сортировки или фильтрации.

ПРИМЕЧАНИЕ В этой главе используются настройки EF Core по умолчанию для связи «многие ко многим». Параметры конфигурации для данного типа связи рассматриваются в главе 8.

ОБНОВЛЕНИЕ СВЯЗИ «МНОГИЕ КО МНОГИМ» ЧЕРЕЗ СВЯЗУЮЩИЙ КЛАСС СУЩНОСТИ

В классе сущности `Book` вам нужна связь «многие ко многим» с сущностями `Author`. Но в книге важен порядок имен авторов. Таким образом, вы создаете связующую таблицу со свойством `Order(byte)`, которое помогает отображать свойства `Name` сущности `Authors` в правильном порядке. Это означает, что вы:

- создаете класс сущности `BookAuthor`, который содержит первичный ключ класса сущности `Book` (`BookId`) и первичный ключ класса сущности `Author` (`AuthorId`). Кроме того, вы добавляете свойство `Order`, которое содержит число, устанавливающее порядок, в котором должны отображаться авторы этой книги. Связующий класс сущности `BookAuthor` также содержит две связи «один к одному» с `Author` и `Book`;
- добавляете навигационное свойство `AuthorsLink` типа `ICollection<BookAuthor>` в класс сущности `Book`;
- добавляете навигационное свойство `BooksLink` типа `ICollection<BookAuthor>` в класс сущности `Author`.

Эти три класса сущностей показаны на рис. 3.5. Видны только связи `Book` с `BookAuthor` и `BookAuthor` с `Author`.

У класса сущности `BookAuthor`, показанного на рис. 3.5, есть два свойства: `BookId` и `AuthorId`. Эти свойства являются внешними ключами для таблиц Books и Authors соответственно. Вместе они также образуют первичный ключ (известный как *составной,* потому что он состоит из нескольких частей) для строки `BookAuthor`. Составной ключ гарантирует, что между `Book` и `Author` только одна связь. Составные ключи подробнее рассматриваются в главе 7. Кроме того, у класса сущности `BookAuthor` также есть свойство `Order`, которое позволяет

определять порядок классов `Author`, чтобы свойство `Name` правильно отображалось в списке книг приложения Book App.

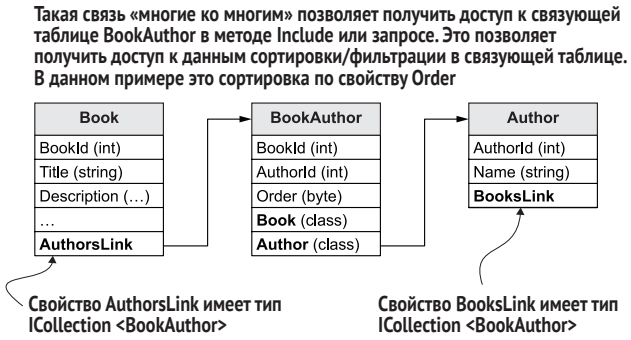


Рис. 3.5 Связь «многие ко многим» книги с ее авторами, где используется связующая таблица `BookAuthor`. Поскольку вы создаете связь «один ко многим» с классом сущности `BookAuthor`, то можете получить доступ к свойству `Order` для сортировки порядка, в котором имена авторов должны быть показаны покупателю

В качестве примера мы добавим автора Мартина Фаулера как дополнительного автора книги «*Quantum Networking*», используя класс `BookAuthor`. (Я уверен, что Мартин Фаулер хотел бы принять участие в работе над этой книгой, если будет жив, когда квантовые сети будут усовершенствованы.) Мы задаем для свойства `Order` значение 1, чтобы сделать Мартина Фаулера вторым автором. (У имеющейся сущности `BookAuthor` для текущего автора у свойства `Order` стоит значение 0.) В следующем листинге показан получившийся в итоге код.

Листинг 3.19 Добавляем нового автора для книги «*Quantum Networking*»

```
var book = context.Books
    .Include(p => p.AuthorsLink)
    .Single(p => p.Title == "Quantum Networking");

var existingAuthor = context.Authors
    .Single(p => p.Name == "Martin Fowler");

book.AuthorsLink.Add(new BookAuthor
{
    Book = book,
    Author = existingAuthor,
    Order = (byte) book.AuthorsLink.Count
});
context.SaveChanges();
```

Мы находим книгу под названием Quantum Networking, автором которой является «человек из будущего»

Находим уже существующего автора – в данном случае это «Мартин Фаулер»

Добавляем новую связующую сущность BookAuthor в коллекцию AuthorsLink

Заполняем два навигационных свойства, которые находятся в связи «многие ко многим»

Устанавливаем для Order значение количества существующих записей AuthorsLink – в данном случае 1 (потому что существует 1 автор и он имеет значение Order = 0)

Метод SaveChanges создаст новую строку в таблице BookAuthor

Следует понимать, что класс сущности `BookAuthor` – это сторона связи *многие*. Листинг, где добавлен еще один автор к одной из книг, должен быть знаком вам, потому что он похож на методы обновления «один ко многим», о которых я уже рассказывал.

Следует отметить, что когда вы загружаете `AuthorsLink`, не нужно загружать соответствующую `BooksLink` в класс сущности `Author`. Причина состоит в том, что когда вы обновляете коллекцию `AuthorsLink`, EF Core знает, что есть связь с `Book`, и во время обновления она автоматически заполнит ее. В следующий раз, когда кто-то загрузит класс сущности `Author` и его связь `BooksLink`, он увидит связь с книгой «*Quantum Networking*» в этой коллекции. (См. раздел 6.2.2, где приводится обзор того, какие ссылки заполняются и когда.)

Также имейте в виду, что удаление записи `AuthorsLink` не приведет к удалению сущностей `Book` или `Author`, с которыми они связаны, потому что эта запись является концом *один* связи «один ко многим», который не зависит от `Book` или `Author`. Фактически классы `Book` или `Author` – это *основные сущности*, причем класс `BookAuthor` зависит от обоих основных сущностей.

ОБНОВЛЕНИЕ СВЯЗИ «МНОГИЕ КО МНОГИМ» С ПРЯМЫМ ДОСТУПОМ К ДРУГОЙ СУЩНОСТИ

В EF Core 5 добавлена возможность доступа к другому классу сущности напрямую в связи «многие ко многим». Эта возможность значительно упрощает настройку и использование данной связи, но вы не сможете получить доступ к связующей таблице в методе `Include`.

> **EF6** В EF6.x можно определить связь «многие ко многим», и EF6.x создаст для вас скрытую связующую таблицу и будет создавать или удалять строки в ней. В EF Core 5 есть такая возможность, но теперь у вас намного больше контроля над конфигурацией связующей таблицы.

В приложении Book App у книги может быть 0 категорий, или их может быть много, например Linux, Базы данных и Microsoft .NET, чтобы помочь покупателю найти нужную книгу. Эти категории находятся в сущности `Tag` (`TagId` содержит название категории) с прямой связью «многие ко многим» с `Book`. Это позволяет `Book` отображать свои категории в списке книг приложения Book App, а также дает возможность приложению предоставлять функцию фильтрации отображаемого списка книг по категории. На рис. 3.6 показаны классы сущностей `Book` и `Tag` и их свойства, которые связываются напрямую друг с другом.

Прямой доступ упрощает добавление или удаление связей между данными сущностями. В следующем листинге показано, как добавить еще один тег для книги «*Quantum Networking*».

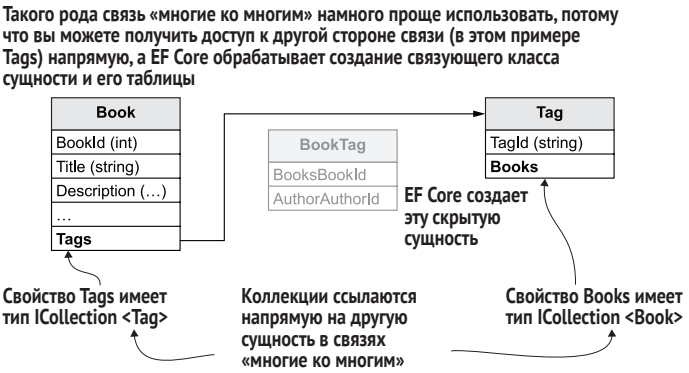


Рис. 3.6 Прямая связь «многие ко многим» между классами сущностей Book и Tag. Вы можете получить доступ к каждому концу связи. EF Core создает скрытый класс сущности, когда видит такого рода связь, и создает правильный код базы данных для использования связанной таблицы

Листинг 3.20 Добавляем тег в книгу, используя прямую связь «многие ко многим»

```
var book = context.Books
    .Include(p => p.Tags)
    .Single(p => p.Title == "Quantum Networking");
```

Находит книгу с названием Quantum Networking и загружает ее с тегами

```
var existingTag = context.Tags
    .Single(p => p.TagId == "Editor's Choice");
```

Находим тег Editor's Choice, чтобы добавить его для этой книги

```
book.Tags.Add(existingTag);
```

Добавляем тег в коллекцию Tags

```
context.SaveChanges();
```

Когда вызывается метод SaveChanges, EF Core создает новую строку в скрытой таблице BookTags

Если вы сравните этот листинг с листингом 3.19, где мы добавили еще одного автора, то увидите, что добавить новую запись в прямую связь «многие ко многим» намного проще. EF Core берет на себя работу по созданию необходимой строки в таблице BookTag. А если бы вы удалили запись в коллекции `Tags`, то удалили бы и соответствующую строку из этой таблицы.

АЛЬТЕРНАТИВНЫЙ СПОСОБ ОБНОВЛЕНИЯ СВЯЗЕЙ: СОЗДАЕМ НОВУЮ СТРОКУ НАПРЯМУЮ

Описав, как обновить два типа связей «многие ко многим», теперь мы обсудим еще один подход: как создать строку связующей таблицы напрямую. Преимущество этого подхода – лучшая производительность, когда у вас много записей в коллекции.

Вместо того чтобы читать коллекцию, можно создать новую запись в связующей таблице. Например, можно создать класс сущности `BookAuthor` и заполнить связи «один к одному» для `Book` и `Author` в этом

классе. Затем вы добавите этот новый экземпляр сущности `BookAuthor` в базу данных и вызовете метод `SaveChanges`. Для коллекции `AuthorsLink`, которая, вероятно, будет небольшой, этот метод, скорее всего, не стоит дополнительных усилий, но в случае со связями «многие ко многим», которые содержат много связанных сущностей, это может значительно улучшить производительность.

3.4.5 Расширенная функция: обновление связей через внешние ключи

Я показал вам, как обновлять связи, используя сами классы сущностей. Например, когда вы добавляли отзыв к книге, то загружали сущность `Book` со всеми ее отзывами. Это нормально, но в отключенном состоянии нужно загружать эту сущность и все ее отзывы по первичному ключу книги, полученному из браузера или REST API. Во многих случаях можно избежать загрузки классов сущностей и вместо этого установить внешние ключи.

Этот метод применим к большинству отключенных обновлений, которые я показывал до сих пор, но позвольте привести пример переноса отзыва из одной книги в другую. (Знаю: в реальном мире такой сценарий маловероятен. Но это просто пример.) В следующем листинге мы выполняем обновление, после того как пользователь напишет отзыв. Предполагается, что `ReviewId` отзыва, который пользователь хочет изменить, и новый `BookId`, который должен быть прикреплен к нему, возвращаются в переменной `dto`.

Листинг 3.21 Обновление внешнего ключа для изменения связи

```
var reviewToChange = context
    .Find<Review>(dto.ReviewId);
reviewToChange.BookId = dto.NewBookId;
context.SaveChanges();
```

Находит отзыв, который вы хотите переместить, используя первичный ключ, возвращенный браузером

Изменяет внешний ключ в отзыве, чтобы он указывал на другую книгу, с которой должен быть связан

Вызывает метод SaveChanges, который обнаруживает, что внешний ключ в отзыве изменен, поэтому он обновляет этот столбец в базе данных

Преимущество этого метода состоит в том, что вам не нужно загружать класс сущности `Book` или использовать команду `Include`, чтобы загрузить все отзывы, связанные с этой книгой. В нашем приложении Book App эти сущности не такие уж большие, но в реальном приложении основная и зависимая сущности могут быть довольно объемными. (Например, у некоторых продуктов Amazon тысячи отзывов.) В отключенных системах, в которых мы часто отправляем только основные ключи, такой подход может быть полезен для сокращения доступа к базе данных, а следовательно, это повысит производительность.

ПРИМЕЧАНИЕ При обновлении связей через внешние ключи может потребоваться доступ к сущностям, у которых нет свой-

ства DbSet<T> в DbContext приложения. Как же читать данные? В листинге 3.21 используется метод Find<T>, но если нужен более сложный запрос, то можно получить доступ к любой сущности с помощью метода Set<T>, например context.Set<Review>().Where(p => p.NumVotes > 5).

3.5 Удаление сущностей

Последний способ изменить данные в базе данных – удалить строку из таблицы. Удалять данные проще, чем выполнять обновления, которые мы уже обсуждали, но здесь есть несколько моментов, о которых нужно знать. Прежде чем я опишу, как удалять сущности из базы данных, я хочу представить подход, называемый *мягким удалением*, когда сущность не удаляется, а скрывается.

ПРИМЕЧАНИЕ Дополнительную информацию об использовании мягкого удаления можно найти в разделе 6.1.7, где приведены определенные ситуации в реальных приложениях.

3.5.1 Мягкое удаление: использование глобального фильтра запросов, чтобы скрыть сущности

Одна из точек зрения гласит, что из базы данных не нужно ничего удалять. Вместо этого нужно использовать состояние, чтобы скрыть данные. Такой подход известен как мягкое удаление (*soft delete*). (См. пост Уди Дахана на странице http://mng.bz/6glD.) Я считаю это разумным, а EF Core предоставляет функцию под названием глобальный фильтр запросов, которая обеспечивает простую реализацию мягкого удаления.

Идея мягкого удаления заключается в том, что в реальных приложениях данные не перестают быть таковыми – они переходят в другое состояние. В случае с нашим примером с книгами книга может еще не поступить в продажу, но факт ее существования не вызывает сомнений, так зачем ее удалять? Вместо этого вы устанавливаете параметр, чтобы сообщить, что сущность нужно скрыть во всех запросах и связях. Чтобы увидеть, как работает этот процесс, мы добавим в список сущностей Book мягкое удаление. Для этого нужно сделать две вещи:

- *добавить логическое свойство* SoftDeleted *в класс сущности* Book. Если это свойство равно true, то экземпляр сущности Book мягко удаляется; его нельзя найти в обычном запросе;
- *добавить глобальный фильтр запросов с помощью текучего* (*fluent*) *API конфигурации EF Core*. Эффект заключается в применении дополнительного фильтра Where к любому доступу к таблице Books.

Добавить свойство `SoftDeleted` в экземпляр сущности `Book` очень просто. В этом фрагменте кода показан класс сущности `Book` со свойством `SoftDeleted`:

```
public class Book
{
    //… Остальные свойства не указаны для ясности;
    public bool SoftDeleted { get; set; }
}
```

Добавление глобального фильтра запроса в свойство `DbSet<Book> Books` означает добавление команды конфигурации EF Core в DbContext приложения. Об этой команде рассказывается в главе 7, а в следующем листинге она выделена жирным шрифтом, чтобы вы имели представление о том, что здесь происходит.

Листинг 3.22 Добавление глобального фильтра запроса к свойству `DbSet<Book>Books`

```
public class EfCoreContext : DbContext
{
    //… Остальные части удалены для наглядности;

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
         //… Остальные части конфигурации удалены для наглядности;

         modelBuilder.Entity<Book>()
             .HasQueryFilter(p => !p.SoftDeleted);
    }
}
```

Добавляет фильтр ко всем запросам к сущностям Book. Можно обойти этот фильтр, используя оператор IgnoreQueryFilters

Чтобы мягко удалить сущность `Book`, необходимо установить для свойства `SoftDeleted` значение `true` и вызвать метод `SaveChanges`. Тогда любой запрос к сущностям `Book` будет исключать сущности `Book`, у которых для свойства `SoftDeleted` установлено значение `true`.

Если вы хотите получить доступ ко всем сущностям с фильтром запросов уровня модели, добавьте к запросу метод `IgnoreQueryFilters`, например `context.Books.IgnoreQueryFilters()`. Этот метод обходит любой фильтр запроса для этой сущности.

> **ПРИМЕЧАНИЕ** Я создал библиотеку под названием `EfCore.SoftDeleteServices`, которая содержит код для настройки и использования этой формы мягкого удаления. Посетите страницу http://mng.bz/op7r для получения дополнительной информации.

Теперь, когда мы рассмотрели подход с мягким удалением, рассмотрим способы реального удаления сущности из базы данных. Начнем с простого примера и перейдем к удалению объекта, имеющего связи.

3.5.2 Удаление только зависимой сущности без связей

Я выбрал класс сущности `PriceOffer`, чтобы продемонстрировать базовое удаление, потому что это зависимая сущность. Следовательно, ее можно удалить, не затрагивая другие сущности. В этом листинге мы находим `PriceOffer`, а затем удаляем ее.

Листинг 3.23 Удаление сущности из базы данных

```
var promotion = context.PriceOffers
    .First();

context.Remove(promotion);
context.SaveChanges();
```

Находит первую сущность PriceOffer

Удаляет ее из DbContext приложения. DbContext определяет, что удалить, в зависимости от типа параметра

Метод SaveChanges вызывает метод DetectChanges, который находит отслеживаемую сущность PriceOffer, помеченную как Deleted, а затем удаляет ее из базы данных

Вызов метода `Remove` устанавливает для свойства `State` сущности, переданной в качестве параметра, значение `Deleted`. Затем, когда вы вызываете метод `SaveChanges`, EF Core находит сущность, помеченную как `Deleted`, и создает правильные команды базы данных для удаления соответствующей строки из таблицы, на которую ссылается сущность (в данном случае это строка в таблице PriceOffers). SQL-команда, которую EF Core создает для SQL Server, показана в следующем фрагменте кода:

```
SET NOCOUNT ON;
DELETE FROM [PriceOffers]
WHERE [PriceOfferId] = @p0;
SELECT @@ROWCOUNT;
```

3.5.3 Удаление основной сущности, у которой есть связи

В разделе 3.3.1 обсуждаются основные и зависимые сущности, а также допустимость значения `null` для внешнего ключа. Реляционные базы данных должны поддерживать *ссылочную целостность*, поэтому если вы удалите строку в таблице, на которую другие строки указывают через внешний ключ, что-то должно будет произойти, чтобы предотвратить потерю ссылочной целостности.

> **ОПРЕДЕЛЕНИЕ** *Ссылочная целостность* – это концепция реляционной базы данных, указывающая, что связи в таблицах всегда должны быть согласованными. Любое поле внешнего ключа должно быть согласовано с первичным ключом, на который ссылается внешний ключ (см. http://mng.bz/XY0M).

Ниже приведены три способа настройки базы данных для сохранения ссылочной целостности при удалении основной сущности с зависимыми сущностями:

- можно указать серверу базы данных удалить зависимые сущности, которые ссылаются на основную сущность. Данный способ известен как *каскадное удаление*;
- можно указать серверу базы данных установить для внешних ключей зависимых сущностей значение `null`, если столбец позволяет это;
- если ни одно из этих правил не настроено, сервер базы данных выдаст ошибку, если вы попытаетесь удалить основную сущность, имеющую зависимые сущности.

3.5.4 Удаление книги с зависимыми связями

В этом разделе мы удалим сущность `Book`, которая представляет собой основную сущность с тремя зависимыми сущностями: `Promotion`, `Reviews` и `AuthorsLink`. Эти сущности не могут существовать без сущности `Book`; внешний ключ, не допускающий значения `null`, связывает эти зависимые сущности с определенной строкой `Book`.

По умолчанию EF Core использует каскадное удаление для зависимых связей с внешними ключами, не допускающими значения `null`. Каскадное удаление упрощает удаление основных сущностей с точки зрения разработчика, потому что два других правила требуют написания дополнительного кода для обработки удаления зависимых сущностей. Но во многих бизнес-приложениях такой подход может оказаться неподходящим. В этой главе используется каскадное удаление, потому что в EF Core это вариант по умолчанию для внешних ключей, не допускающих значения `null`.

Помня об этом, посмотрим, как работает каскадное удаление, используя настройку по умолчанию для удаления сущности `Book`, у которой есть связи. В этом листинге мы загружаем связи `Promotion` (класс сущности `PriceOffer`), `Reviews`, `AuthorsLink` и `Tags` с классом сущности `Book`, перед тем как удалить эту сущность.

Листинг 3.24 Удаление книги с тремя зависимыми классами сущностей

```
var book = context.Books
    .Include(p => p.Promotion)
    .Include(p => p.Reviews)
    .Include(p => p.AuthorsLink)
    .Include(p => p.Tags)
    .Single(p => p.Title
        == "Quantum Networking");
context.Books.Remove(book);
context.SaveChanges();
```

Четыре метода Include гарантируют, что четыре зависимые связи загружены с Book

Находит книгу Quantum Networking, у которой, как вы знаете, есть рекламная акция, два отзыва, одна связь с BookAuthor и одна с BookTag

Удаляет эту книгу

Метод SaveChanges вызывает метод DetectChanges, который находит отслеживаемую сущность Book, помеченную как Deleted, удаляет ее зависимые связи, а затем удаляет книгу

Мои тестовые данные содержат книгу под названием «*Quantum Networking*», в которой есть одна сущность `PriceOffer`, две сущности `Review` и сущность `BookAuthor`. Внешние ключи всех этих зависимых сущностей, которые я упомянул, указывают на книгу «*Quantum Networking*». После выполнения кода из листинга 3.24 EF Core удаляет `Book`, `PriceOffer`, две сущности `Review`, одну `BookAuthor` и одну (скрытую) `BookTag`.

Последнее утверждение, указывающее, что все они удаляются EF Core, – важный момент. Поскольку вы добавили четыре метода `Include`, EF Core знал о зависимых сущностях и выполнил удаление. Если бы вы не включили эти методы в свой код, EF Core не знал бы о зависимых сущностях и не смог бы удалить три зависимые сущности. В этом случае проблема сохранения ссылочной целостности будет лежать на сервере базы данных, и его ответ будет зависеть от того, как была настроена часть `ON DELETE` ограничения по внешнему ключу. Базы данных, созданные EF Core для этих классов сущностей, по умолчанию будут настроены на использование каскадного удаления.

> **ПРИМЕЧАНИЕ** `Author` и `Tag`, связанные с `Book`, не удаляются, потому что они не являются зависимыми сущностями `Book`; удаляются только сущности `BookAuthor` и `BookTag`. Это не лишено смысла, потому что `Author` и `Tag` можно использовать и с другими сущностями `Book`.

В разделе 8.8.1 показано, как настроить способ, которым EF Core удаляет зависимую сущность в связях. Иногда полезно не удалять основную сущность, если с ней связана определенная зависимая сущность. В нашем приложении Book App, например, если клиент заказывает книгу, вы хотите сохранить эту информацию о заказе, даже если книга больше не продается. В этом случае вы меняете действие on-delete на `Restrict` и удаляете `ON DELETE CASCADE` из ограничения по внешнему ключу в базе данных, чтобы при попытке удалить книгу была выдана ошибка.

> **ПРИМЕЧАНИЕ** Когда вы удаляете основную сущность с зависимой сущностью, у которой есть внешний ключ, допускающий значение `null` (известно как *необязательная зависимая связь*), есть тонкие различия между тем, как EF Core выполняет удаление, и тем, как это делает база данных. Я объясняю эту ситуацию в разделе 8.8.1, используя полезную табл. 8.1.

Резюме

- У экземпляров сущностей есть состояние (свойство `State`), которое может принимать значения `Added`, `Unchanged`, `Modified`, `Deleted` или

`Detached`. Состояние определяет, что происходит с сущностью при вызове метода `SaveChanges`.

- Если вы добавляете сущность, то для свойства `State` устанавливается значение `Added`. При вызове метода `SaveChanges` эта сущность записывается в базу данных в виде новой строки.
- Вы можете обновить одно или несколько свойств в классе сущности, загрузив класс сущности как отслеживаемую сущность, изменив свойства и вызвав метод `SaveChanges`.
- В реальных приложениях используются два типа сценариев обновления – с подключенным и отключенным состояниями: это влияет на способ выполнения обновления.
- В EF Core есть метод `Update`, который помечает весь класс сущности как обновленный. Можно использовать этот метод, когда вы хотите обновить класс сущности и все данные сущности уже вам доступны.
- При обновлении связи есть два варианта, с разными преимуществами и недостатками:
  - можно загрузить существующую связь с основной сущностью и обновить эту связь в основной сущности. Дальше EF Core разберется сам. Этот вариант проще использовать, но он может создать проблемы с производительностью, когда вы имеете дело с большими коллекциями данных;
  - можно создать, обновить или удалить зависимую сущность. Такой подход труднее понять, но обычно он быстрее, потому что не нужно загружать существующие связи.
- Чтобы удалить сущность из базы данных, используется метод `Remove`, за которым следует метод `SaveChanges`.

Для читателей, знакомых с EF6.x:

- метод `Update` – это долгожданная новая команда EF Core. В EF6.x для этого нужно использовать `DbContext.Entry(object).State`;
- EF Core предоставляет сокращенные варианты `Add`, `Update` и `Remove`. Вы можете применить любую из этих команд к самому контексту, как в `context.Add(book)`;
- в EF6.x по умолчанию метод `SaveChanges` проверяет данные перед добавлением или обновлением сущности в базе данных. EF Core не выполняет никаких проверок для `SaveChanges`, но их легко добавить (см. главу 4);
- EF6.x позволяет напрямую определять связь «многие ко многим» и заботится о создании связующей таблицы и управлении строками, чтобы этот процесс работал. NET Core 5 добавляет эту функцию в EF Core; раздел 3.4.4 посвящен данной теме.

4 Использование EF Core в бизнес-логике

В этой главе рассматриваются следующие темы:

- бизнес-логика и ее использование в EF Core;
- три типа бизнес-логики, от простой к сложной;
- обзор каждого типа бизнес-логики, их плюсы и минусы;
- добавление шага, который проверяет данные, перед тем как они будут записаны в базу данных;
- использование транзакций для объединения строк кода в одну логическую цепочку.

Реальные приложения созданы для предоставления набора услуг, от хранения простого списка вещей на вашем компьютере до управления ядерным реактором. У каждой реальной проблемы есть набор правил, часто называемых *бизнес-правилами*, или *правилами предметной области*. (В этой книге используется термин *бизнес-правила*.)

Код, который вы пишете для реализации бизнес-правила, известен как *бизнес-логика*, или *логика предметной области*. Поскольку бизнес-правила могут быть сложными, написанная вами бизнес-логика также может быть сложной. Просто задумайтесь обо всех проверках и этапах, которые необходимо выполнить, когда вы заказываете что-то онлайн.

Бизнес-логика может варьироваться от простой проверки состояния до массивного кода искусственного интеллекта, но почти во всех

случаях бизнес-логике требуется доступ к базе данных. Хотя все подходы, описанные в главах 2 и 3, работают, способ применения этих команд EF Core для бизнес-логики может немного отличаться, поэтому я и написал эту главу.

В ней описывается паттерн для обработки бизнес-логики, который разделяет некоторые сложные аспекты на части, чтобы снизить нагрузку на вас, разработчиков. Вы также узнаете несколько методов написания различных типов бизнес-логики, использующих EF Core для доступа к базе данных. Эти методы варьируются от использования программных классов для валидации до стандартизации интерфейса бизнес-логики с целью упрощения кода внешнего интерфейса. Цель главы – помочь вам быстро написать точную, понятную и правильно работающую бизнес-логику.

4.1 Вопросы, которые нужно задать, и решения, которые нужно принять, прежде чем начать писать код

Наш код в главах 2 и 3, где используются CRUD-операции, адаптировал и преобразовывал данные по мере их перемещения в базу данных и обратно. Часть этого кода была сложной, и я показал вам паттерн «Объект запроса», чтобы сделать большой запрос более управляемым. Точно так же бизнес-логика может быть простой или сложной.

> **ОПРЕДЕЛЕНИЕ** В этой главе я использую термин *бизнес-правило*, чтобы представить удобную для восприятия формулировку логики, которую необходимо реализовать, например «цена книги не может быть отрицательной». Кроме того, я использую термин *бизнес-логика*, то есть код, реализующий все бизнес-правила, необходимые для определенной функции в приложении.

Прежде чем приступить к работе над своей бизнес-логикой, следует подумать над ответами на следующие вопросы:

- понимаете ли вы бизнес-правила реализуемой функциональности?
- имеют ли бизнес-правила смысл или они неполные?
- есть ли какие-то крайние случаи или исключения, которые нужно учесть?
- как доказать, что ваша реализация соответствует бизнес-правилам?
- насколько легко будет изменить код, если изменятся бизнес-правила?

4.1.1 Три уровня сложности кода бизнес-логики

Если вы понимаете бизнес-правила, которые необходимо реализовать, то должны иметь представление о том, насколько сложна бизнес-логика. Большинство правил будет просто реализовать, а некоторые будут действительно сложными. Хитрость состоит в том, чтобы быстро реализовать простую бизнес-логику, но использовать более структурированный подход для сложной бизнес-логики.

Основываясь на собственном опыте, я составил список из трех уровней сложности бизнес-логики с разными шаблонами для каждого уровня: проверка, простая и сложная бизнес-логика.

В следующих трех разделах описаны эти уровни и их влияние на код, который вы пишете. Но имейте в виду, что эти три шаблона не являются строгими правилами.

Некоторые бизнес-правила могут быть простыми, но, возможно, вы решите использовать более сложный шаблон, потому что его проще тестировать. Тем не менее этот список полезен для обсуждения типов и шаблонов, которые можно использовать для написания кода бизнес-логики.

КОД ВАЛИДАЦИИ ДЛЯ ПРОВЕРКИ ДАННЫХ, ИСПОЛЬЗУЕМЫХ ДЛЯ ИЗМЕНЕНИЯ КЛАССА СУЩНОСТИ

При работе с операциями CUD (create, update и delete), как в главе 3, вам может потребоваться проверить, находятся ли данные в определенном диапазоне. Например, свойство `NumStars` сущности `Review` должно находиться в диапазоне от 0 до 5. Данный вид теста известен как *валидация*. Для меня валидация – это отправная точка, чтобы назвать код *бизнес-логикой* вместо *кода операций CRUD*.

Это распространенный тип бизнес-логики; его можно встретить повсюду (см. врезку «Должен ли код бизнес-логики находиться только в специально выделенном слое?» перед разделом 4.2). При простейшей валидации обычно используются операторы `if-then`, которые проверяют значения данных, но полезный набор атрибутов, Data Annotations, может автоматизировать часть кода валидации, который вам нужно писать. (С Data Annotations вы познакомитесь позже, в разделе 4.7.1.)

Существует множество уровней валидации, от простой проверки диапазона до проверки действительности водительских прав с помощью некоего сервиса, что затрудняет определение границ начального уровня бизнес-логики. Но, как я сказал вначале, эти уровни являются руководящими принципами, и пример проверки «водительских прав» поднимает этот код на следующий уровень бизнес-логики.

ПРОСТАЯ БИЗНЕС-ЛОГИКА (НЕМНОГО ВЕТВЛЕНИЯ ИЛИ ПОЛНОЕ ЕГО ОТСУТСТВИЕ И ЛЕГКОСТЬ ВОСПРИЯТИЯ)

Следующий тип – это бизнес-логика, в которой ветвление практически отсутствует, т. е. здесь есть небольшое количество операторов

ветвления `if-then` или их нет вообще и нет обращения к другому бизнес-коду. Данный код легок для восприятия, потому что вы можете прочитать его и увидеть каждый шаг, который должен быть выполнен по порядку. Хороший пример – код для создания книги с авторами, требующий создания `Book`, поиска или создания авторов и, наконец, добавления связующих классов сущностей, `BookAuthor`. Это простой код, где нет ветвлений, но все же здесь требуется много строк, чтобы создать книгу с авторами.

Меня всегда удивляет, сколько такой «простой» бизнес-логики присутствует в реальном приложении; как правило, я считаю, что в эту категорию попадает множество функций администратора. Поэтому наличие простого шаблона для построения и проверки данного типа бизнес-логики имеет решающее значение для быстрого создания кода.

СЛОЖНАЯ БИЗНЕС-ЛОГИКА (КОД, ТРЕБУЮЩИЙ СЕРЬЕЗНЫХ УСИЛИЙ, ЧТОБЫ НАПИСАТЬ ЕГО ПРАВИЛЬНО)

Самую трудную для написания бизнес-логику я называю *сложной*. У этого термина нет подходящего определения, но для данного типа кода нужно хорошо подумать над проблемой, прежде чем ее можно будет реализовать. Вот цитата из одной из ведущих книг по написанию бизнес-логики, в которой описывается задача написания кода сложной бизнес-логики:

> *В алгоритмической части программы сосредоточена ее способность решать для пользователя задачи из соответствующей предметной области. Все остальные части программы и ее функции, какими бы существенными они ни были, только служат этой основной цели. Если область сложна, то и решение соответствующих задач – дело непростое, требующее сосредоточенных усилий талантливых и умелых специалистов.*
>
> – Эрик Эванс,
> «Предметно-ориентированное проектирование»[1]

Этот тип бизнес-логики достаточно сложен, поэтому я разработал структурированный подход, изолирующий бизнес-логику от базы данных и клиентской части. Таким образом, я могу сосредоточиться на проблеме бизнес-логики – еще одном применении принципа разделения ответственностей (о котором подробно рассказывается в разделе 5.5.2).

---

[1] *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003).

Должен ли код бизнес-логики находиться только в специально выделенном слое?

Нет. В реальных приложениях, особенно в тех, которые взаимодействуют с человеком, необходимо, чтобы это взаимодействие было как можно лучше. По этой причине некоторая бизнес-логика находится в слое представления.

Очевидная логика для размещения в слое представления – это бизнес-логика проверки, потому что чем раньше вы сможете предоставить обратную связь пользователю, тем лучше. Большинство frontend-систем имеют встроенные возможности, облегчающие проверку и предоставляющие пользователю информацию об ошибках.

Еще одна область – это бизнес-логика, состоящая из множества этапов. Часто для пользователя лучше, когда потоки сложной бизнес-логики отображаются в виде последовательности страниц или этапов в мастере.

Даже в серверной части приложения я распределяю бизнес-логику на несколько слоев (проектов) в приложении Book App. В этой главе я объясняю, как и почему это делаю.

4.2 Пример сложной бизнес-логики: обработка заказа на приобретение книги

Я начинаю со сложной бизнес-логики, потому что она познакомит вас с мощным подходом, взятым из книги Эрика Эванса «*Предметно-ориентированное проектирование*», которую я цитирую в предыдущем разделе. Однако для начала взгляните на функциональность сложной бизнес-логики, которую нужно будет реализовать в приложении Book App. Пример, который мы создадим, – это обработка пользовательского заказа на приобретение книг. На рис. 4.1 показана страница оформления заказа в приложении Book App. Мы хотим реализовать код, который выполняется, когда **пользователь нажимает кнопку «Купить»**.

ПРИМЕЧАНИЕ Можно опробовать процесс оформления заказа, скачав код приложения Book App из связанного репозитория Git и запустив его локально. Приложение использует HTTP cookie для хранения информации о корзине и пользователе (что избавляет от необходимости выполнять вход в систему). Деньги не нужны; как сказано в тексте условий использования, на самом деле мы не собираемся покупать книгу.

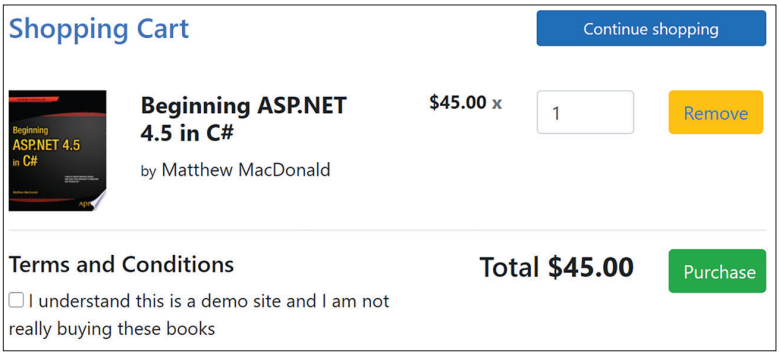


Рис. 4.1 Страница оформления заказа в приложении Book App. Когда пользователь нажимает кнопку «Купить книгу» рядом с книгой, приложение добавляет книгу в корзину, после чего отображается страница оформления заказа, на которой показаны все книги в корзине пользователя. Нажимая кнопку «Купить», вы вызываете бизнес-логику, которая создает заказ, т. е. код, который мы будем писать

4.3 Использование паттерна проектирования для реализации сложной бизнес-логики

Прежде чем начать писать код для обработки заказа, взгляните на паттерн, который поможет вам написать и протестировать бизнес-логику, а также настроить ее производительность. Он основан на концепциях предметно-ориентированного проектирования (*DDD*), изложенных Эриком Эвансом, но код бизнес-логики не находится внутри классов сущностей. Этот паттерн известен как *сценарий транзакции*, или *процедурный* паттерн бизнес-логики, потому что код содержится в автономном методе.

Он прост для понимания и использует базовые команды EF Core, которые вы уже видели. Но многие рассматривают процедурный подход как антипаттерн DDD, известный как *анемичная модель предметной области* (см. http://mng.bz/nM7g). Позже, в части III книги, мы расширим этот подход до полноценного DDD.

В этом разделе и в главе 13 представлена моя интерпретация подхода Эванса и множество других способов применения DDD с EF. Хотя я предлагаю свой подход, который, надеюсь, поможет вам, не бойтесь искать другие подходы.

4.3.1 Пять правил по созданию бизнес-логики, использующей EF Core

В следующем списке приводятся пять правил, составляющих паттерн бизнес-логики, который вы будете использовать в этой главе. Большая

их часть берет свое начало из концепций DDD, а некоторые являются результатом написания большого количества сложной бизнес-логики и выявления областей, которые нужно улучшить:

- *Бизнес-логика определяет структуру базы данных*. Поскольку проблема, которую вы пытаетесь решить (которую Эванс называет *моделью предметной области*), – это сердце системы, ее логика должна определять способ проектирования всего приложения. Поэтому вы пытаетесь сделать так, чтобы структура базы данных и классы сущностей максимально соответствовали потребностям данных бизнес-логики.
- *Ничто не должно отвлекать от бизнес-логики*. Само по себе написание бизнес-логики – достаточно сложный процесс, поэтому вы изолируете ее от всех других слоев приложения, кроме классов сущностей. Когда вы пишете код бизнес-логики, то должны думать только о проблеме, которую пытаетесь решить. Вы оставляете задачу адаптации данных для представления сервисному слою приложения.
- *Бизнес-логика должна думать, что работает с данными в памяти*. Эванс научил меня писать код бизнес-логики так, будто данные находятся в памяти. Конечно, нужно иметь какие-то части, касающиеся *загрузки* и *сохранения*, но что касается ядра вашей бизнес-логики, обрабатывайте данные (насколько это возможно), как если бы это был обычный класс или коллекция в памяти.
- *Изолируйте код доступа к базе данных в отдельном проекте*. Это правило появилось в результате написания приложения для электронной коммерции со сложными правилами ценообразования и доставки. Раньше я использовал EF непосредственно в своей бизнес-логике, но обнаружил, что она сложна в сопровождении и с точки зрения настройки производительности. Вместо этого нужно использовать другой проект, дополняющий бизнес-логику, для хранения всего кода доступа к базе данных.
- *Бизнес-логика не должна вызывать метод EF Core* `SaveChanges` *напрямую*. У вас должен быть класс в сервисном слое (или пользовательская библиотека), задача которого – вызывать бизнес-логику. Если ошибок нет, данный класс вызывает метод `SaveChanges`. Основная причина этого правила – контролировать, записывать ли данные, но у него есть и другие преимущества, которые я опишу в разделе 4.4.5.

На рис. 4.2 показана структура приложения, которую мы создадим, что поможет вам в применении этих рекомендаций при реализации бизнес-логики. В нашем случае мы добавим два новых проекта к исходной структуре приложения Book App, описанной в главе 2:

- проект с чистой бизнес-логикой, содержащий классы бизнес-логики, которые работают с данными в памяти, предоставляемыми сопутствующими методами доступа к базе данных;

- проект доступа к базе данных, предоставляющий сопутствующий класс для каждого класса чистой бизнес-логики, которому требуется доступ к базе данных. Каждый сопутствующий класс заставляет класс чистой бизнес-логики думать, что он работает с набором данных в памяти.

Пять номеров с комментариями на рис. 4.2 соответствуют пяти правилам.

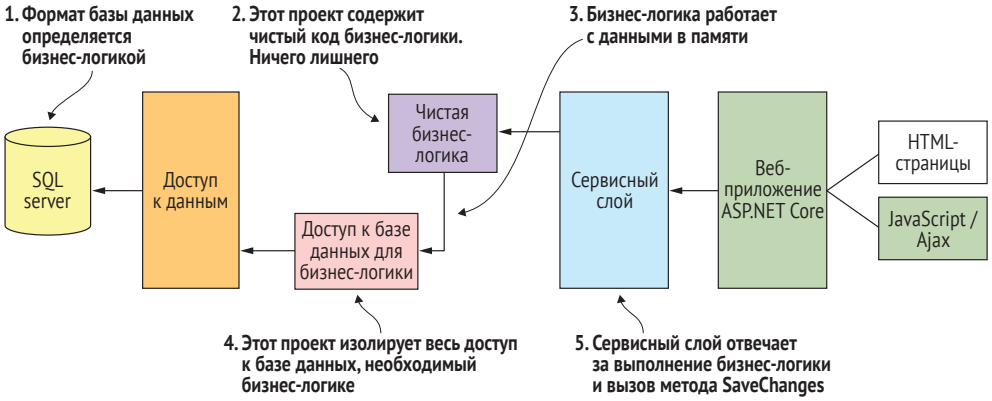


Рис. 4.2 Проекты внутри нашего приложения Book App с двумя новыми проектами для обработки сложной бизнес-логики. Проект «Чистая бизнес-логика» содержит изолированную бизнес-логику, которая считает, что работает с набором классов в памяти. Проект «Доступ к базе данных» предоставляет интерфейс, который чистая бизнес-логика может использовать для доступа к базе данных. Задача сервисного слоя – адаптировать данные из приложения ASP.NET Core для отправки в чистую бизнес-логику в нужной форме и вызывать окончательный метод SaveChanges для сохранения, если бизнес-логика не сообщает об ошибках

4.4 Реализация бизнес-логики для обработки заказа

Теперь, когда я описал бизнес-потребность, ее бизнес-правила и паттерн, который мы будем использовать, можно приступать к написанию кода. Наша цель состоит в том, чтобы разбить реализацию на более мелкие шаги, которые сосредоточены на конкретных частях проблемы. Вы увидите, как этот паттерн бизнес-логики помогает сосредоточиться на каждой части реализации по очереди.

Мы собираемся реализовать код по частям, которые соответствуют пяти правилам, перечисленным в разделе 4.3.1. В конце вы увидите, как этот код вызывается из ASP.NET Core, который использует приложение Book App.

4.4.1 Правило 1: бизнес-логика требует определения структуры базы данных

В нем говорится, что проектирование базы данных должно соответствовать бизнес-потребностям – в данном случае они представлены шестью бизнес-правилами. Только три из них имеют отношение к проектированию базы данных:

- в заказ должна быть включена хотя бы одна книга (подразумевается, что их может быть больше);
- цена книги должна быть скопирована в заказ, потому что позже она может поменяться;
- нужно запомнить человека, заказавшего книги.

Эти три правила определяют класс сущности `Order`, у которого есть коллекция классов сущностей `LineItem` – связь «один ко многим». Класс сущности `Order` содержит информацию о человеке, разместившем заказ, а каждый экземпляр `LineItem` содержит ссылку на заказ книги, количество книг и стоимость.

На рис. 4.3 показано, как эти две таблицы, LineItem и Orders, выглядят в базе данных. Чтобы изображение было более понятным, я показываю таблицу Books (бледно-серого цвета), на которую ссылается каждая строка LineItem.

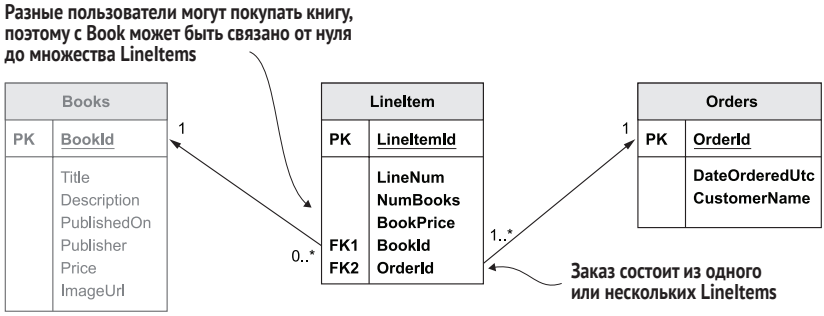


Рис. 4.3 Добавлены новые таблицы LineItem и Orders, позволяющие принимать заказы на книги. На каждую покупку есть одна строка Orders, со строкой LineItem для каждой книги

ПРИМЕЧАНИЕ Имя таблицы Orders стоит во множественном числе, потому что мы добавили `DbSet<Order>Orders` в DbContext приложения, и по умолчанию EF Core использует имя свойства Orders в качестве имени таблицы. Мы не добавили свойство для класса сущности `LineItem`, поскольку доступ к нему осуществляется через связь Order. В этом случае EF Core по умолчанию использует имя класса `LineItem` в качестве имени таблицы. Можно задать для таблицы определенное имя; см. раздел 7.11.1.

4.4.2 Правило 2: ничто не должно отвлекать от бизнес-логики

Теперь вы находитесь в сердце кода бизнес-логики, и код здесь сделает большую часть работы. Он будет самой сложной частью реализации, которую вы пишете, но нам нужно упростить задачу, избавившись от всех отвлекающих факторов. Так мы сможем сосредоточиться на проблеме.

Для этого нужно писать чистый бизнес-код со ссылкой только на две другие части системы: классы сущностей, показанные на рис. 4.3 (`Order`, `LineItem` и `Book`), и сопутствующий класс, который будет обрабатывать все обращения к базе данных. Даже при такой минимизации вы все равно будете разбивать работу на несколько частей.

ПРОВЕРКА ОШИБОК И ПЕРЕДАЧА ИХ ПОЛЬЗОВАТЕЛЮ: ВАЛИДАЦИЯ

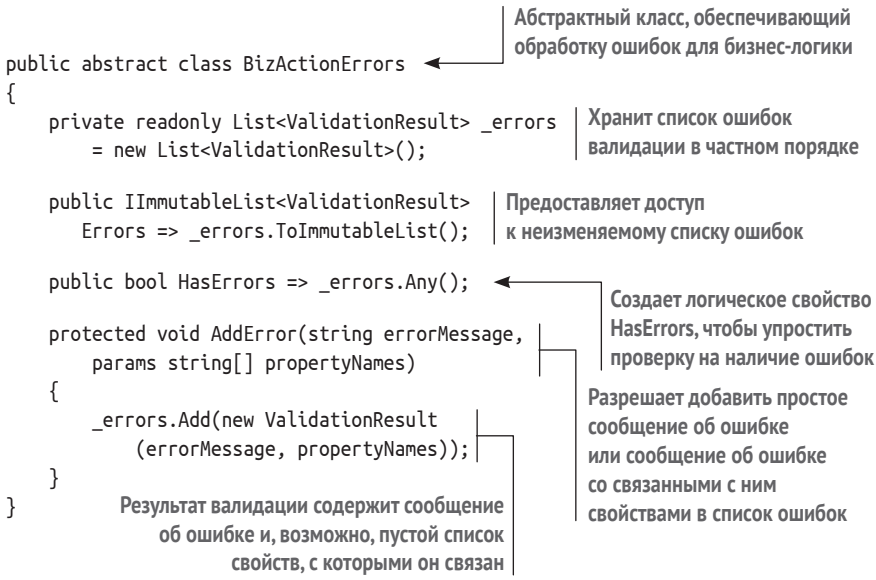
Бизнес-правила содержат несколько флажков, например «Необходимо поставить галочку напротив поля “Пользовательское соглашение”». В правилах также говорится, что нужно предоставить пользователю механизм обратной связи, чтобы он мог исправить любые проблемы и завершить покупку. Такого рода проверки, называемые *валидацией*, распространены во всем приложении.

Есть два основных подхода к обработке передачи ошибок на более высокие слои. Один из них – генерировать исключение при возникновении ошибки, а другой – передавать ошибки вызывающей стороне через интерфейс состояния. У каждого варианта есть свои достоинства и недостатки. В этом примере используется второй подход: передача ошибок в форме класса состояния в более высокий слой для проверки.

Для этого мы создадим небольшой абстрактный класс `BizActionErrors`, показанный в листинге 4.1. Этот класс предоставляет общий интерфейс обработки ошибок для всей вашей бизнес-логики. Класс содержит метод `AddError`, который бизнес-логика может вызывать для добавления ошибки, и *неизменяемый список* `Errors`, содержащий все ошибки валидации, обнаруженные при выполнении бизнес-логики.

Мы будем использовать класс `ValidationResult` для хранения каждой ошибки, потому что это стандартный способ возврата ошибок с необязательной дополнительной информацией о точном свойстве, с которым связана ошибка. Использование этого класса вместо простой строки соответствует еще одному методу валидации, который мы добавим позже в данной главе.

Листинг 4.1 Абстрактный базовый класс, обеспечивающий обработку ошибок для вашей бизнес-логики




```
public abstract class BizActionErrors
{
    private readonly List<ValidationResult> _errors
        = new List<ValidationResult>();

    public IImmutableList<ValidationResult>
       Errors => _errors.ToImmutableList();

    public bool HasErrors => _errors.Any();

    protected void AddError(string errorMessage,
        params string[] propertyNames)
    {
        _errors.Add(new ValidationResult
            (errorMessage, propertyNames));
    }
}
```

Использование этого абстрактного класса означает, что код вашей бизнес-логики легче писать и у нее имеется единый способ обработки ошибок. Другое преимущество заключается в том, что можно изменить способ внутренней обработки ошибок без изменения кода бизнес-логики.

Бизнес-логика обработки заказа выполняет много проверок, что типично для заказа, потому что это часто связано с деньгами. Другая бизнес-логика может не выполнять никакой проверки, но базовый класс `BizActionErrors` автоматически возвращает `HasErrors` со значением `false`, а это означает, что со всей бизнес-логикой можно работать одинаково.

4.4.3 Правило 3: бизнес-логика должна думать, что работает с данными в памяти

Теперь начнем с основного класса: `PlaceOrderAction`, который содержит чистую бизнес-логику. Он использует сопутствующий класс `PlaceOrderDbAccess` для представления данных в виде набора в памяти (в данном случае словаря) и для записи созданного заказа в базу данных. Хотя мы не пытаемся скрыть базу данных от чистой бизнес-логики, нам нужно, чтобы она работала так, как если бы данные были обычными классами .NET.

В листинге 4.2 показан класс `PlaceOrderAction`, наследующий от абстрактного класса `BizActionErrors` для обработки сообщений об ошибках, возвращаемых пользователю. Кроме того, он использует два метода, которые предоставляет сопутствующий класс `PlaceOrderDbAccess`:

- `FindBooksByIdsWithPriceOffers` – принимает список `BookId` и возвращает словарь с `BookId` в качестве ключа и класс сущности `Book` в качестве значения и любых связанных `PriceOffers`;
- `Add` – добавляет класс сущности `Order` с его коллекцией `LineItem` в базу данных.

Листинг 4.2 Класс `PlaceOrderAction` с бизнес-логикой, используемой для построения нового заказа

```
public class PlaceOrderAction :
    BizActionErrors,
    IBizAction<PlaceOrderInDto, Order>
{
    private readonly IPlaceOrderDbAccess _dbAccess;

    public PlaceOrderAction(IPlaceOrderDbAccess dbAccess)
    {
       _dbAccess = dbAccess;
    }

    public Order Action(PlaceOrderInDto dto)
    {
        if (!dto.AcceptTAndCs)
        {
            AddError(
"You must accept the T&Cs to place an order.");
            return null;
        }
        if (!dto.LineItems.Any())
        {
            AddError("No items in your basket.");
            return null;
        }

        var booksDict =
            _dbAccess.FindBooksByIdsWithPriceOffers
                 (dto.LineItems.Select(x => x.BookId));
        var order = new Order
        {
            CustomerId = dto.UserId,
            LineItems =
                FormLineItemsWithErrorChecking
                     (dto.LineItems, booksDict)
        };

        if (!HasErrors)
            _dbAccess.Add(order);
```

Класс BizActionErrors обеспечивает обработку ошибок для бизнес-логики

Интерфейс IBizAction обеспечивает соответствие бизнес-логики стандартному интерфейсу

PlaceOrderAction использует класс PlaceOrderDbAccess для обработки обращений к базе данных

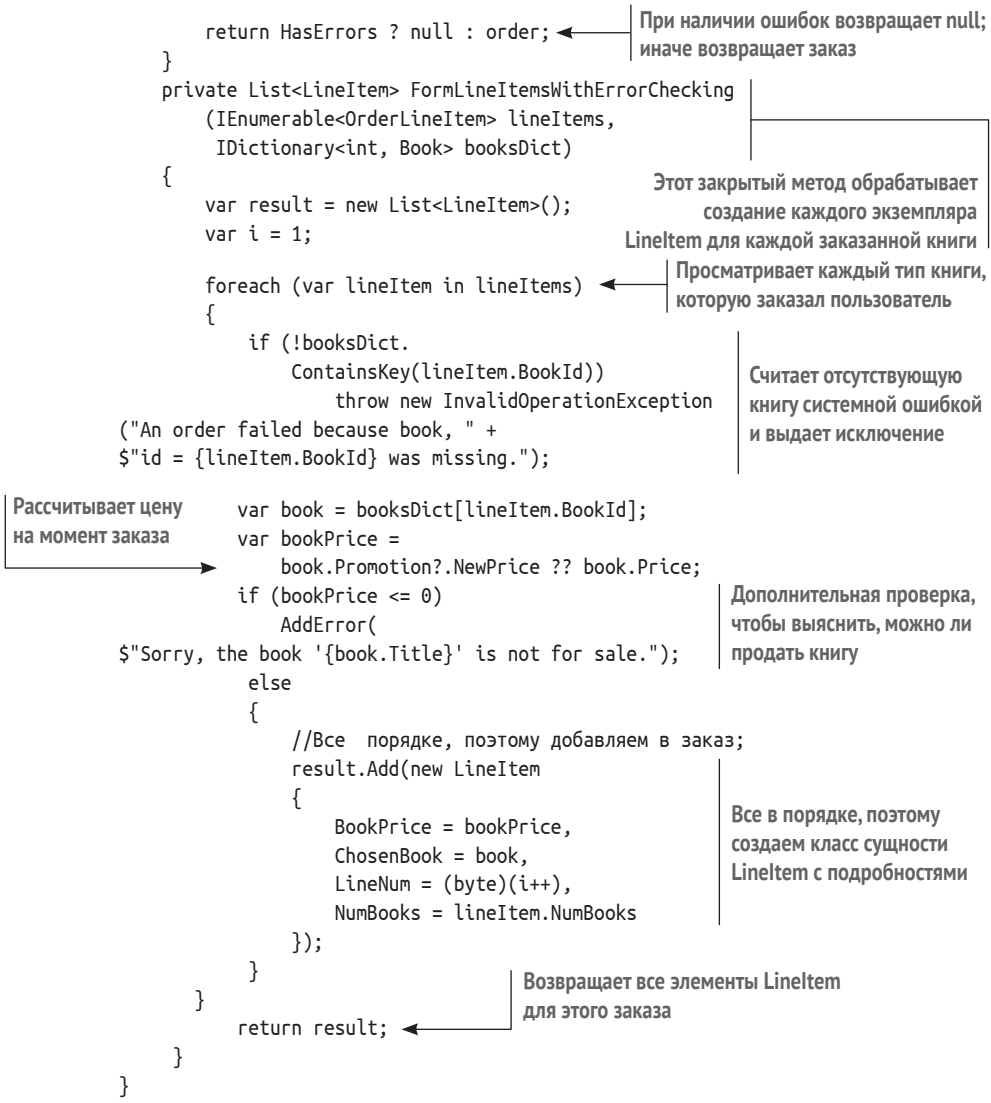
BizRunner вызывает этот метод для выполнения бизнес-логики

Базовая проверка

Класс PlaceOrderDbAccess находит все купленные книги с дополнительными PriceOffers

Создает заказ, используя FormLineItemsWithErrorChecking для создания LineItems

Добавляет заказ в базу только при отсутствии ошибок



Вы можете заметить, что добавлена еще одна проверка того, что книга, выбранная пользователем, все еще находится в базе данных. Этой проверки не было в бизнес-правилах, но она имеет смысл, особенно если пользователем умышленно предоставлены неверные данные. В этом случае делается различие между ошибками, которые может исправить пользователь, которые возвращаются свойством `Errors`, и системными ошибками (в данном случае отсутствующая книга), для которых вы генерируете исключение, которое система должна регистрировать.

Возможно, вы видели в верхней части класса, что мы используем интерфейс `IBizAction<PlaceOrderInDto,Order>`. Это гарантирует, что класс соответствует стандартному интерфейсу, который вы исполь-

зуете во всей бизнес-логике. Вы увидите это в разделе 4.7.1, когда мы создадим обобщенный класс для запуска и проверки бизнес-логики.

4.4.4 Правило 4: изолируйте код доступа к базе данных в отдельный проект

В наших правилах сказано, что весь код доступа к базе данных, необходимый для бизнес-логики, следует поместить в отдельный сопутствующий класс. Этот метод гарантирует, что все обращения к базе данных находятся в одном месте, что значительно упрощает тестирование, рефакторинг и настройку производительности.

Еще одно преимущество, которое заметил читатель моего блога, заключается в том, что это руководство может помочь, если вы работаете с существующей старой базой данных. В этом случае сущности из базы данных могут не подходить для кода бизнес-логики, который вы хотите написать. Тогда можно использовать методы `BizDbAccess` в качестве паттерна *Адаптер*, который преобразует старую структуру базы данных в форму, с которой вашей бизнес-логике легче работать.

> **ОПРЕДЕЛЕНИЕ** Паттерн *Адаптер* преобразует интерфейс одного класса в другой интерфейс, которого ожидает клиент. Этот паттерн позволяет классам работать вместе, что было бы невозможно из-за несовместимых интерфейсов. Посетите страницу https://sourcemaking.com/design_patterns/adapter для получения дополнительной информации.

Убедитесь, что ваша чистая бизнес-логика, класс `PlaceOrderAction` и класс доступа к базе данных `PlaceOrderDbAccess` находятся в отдельных проектах. Такой подход позволяет исключить любые библиотеки EF Core из проекта чистой бизнес-логики, гарантируя, что весь доступ к базе данных осуществляется через сопутствующий класс `PlaceOrderDbAccess`. В своих проектах я выделяю классы сущностей в отдельный проект без зависимости от кода EF. Тогда в моем проекте чистой бизнес-логики не будет библиотеки NuGet `Microsoft.EntityFrameworkCore`, поэтому моя бизнес-логика не сможет выполнять какие-либо команды базы данных напрямую; для доступа к данным она должна полагаться на класс `PlaceOrderDbAccess`.

Для простоты пример кода содержит классы сущностей в том же проекте, что и DbContext приложения. В листинге 4.3 показан наш класс `PlaceOrderDbAccess`, реализующий два метода для обеспечения доступа к базе данных, необходимых чистой бизнес-логике:

- метод `FindBooksByIdsWithPriceOffers`, который находит и загружает каждый класс сущности `Book` с дополнительным `PriceOffer`;
- метод `Add`, добавляющий готовый класс сущности `Order` к свойству DbContext приложения, `Orders`, чтобы его можно было сохранить в базе данных после вызова метода `SaveChanges`.

Листинг 4.3 Класс PlaceOrderDbAccess, обрабатывающий все обращения к базе данных

```
public class PlaceOrderDbAccess : IPlaceOrderDbAccess
{
    private readonly EfCoreContext _context;

    public PlaceOrderDbAccess(EfCoreContext context)
    {
        _context = context;
    }

    public IDictionary<int, Book>
        FindBooksByIdsWithPriceOffers
            (IEnumerable<int> bookIds)
    {
       return _context.Books
           .Where(x => bookIds.Contains(x.BookId))
           .Include(r => r.Promotion)
           .ToDictionary(key => key.BookId);
    }

    public void Add(Order newOrder)
    {
        _context.Add(newOrder);
    }
}
```

Для всех BizDbAccess требуется DbContext для доступа к базе данных

Находит все книги, которые хочет купить пользователь

BizLogic получает коллекцию BookId, предоставленную пользователем на этапе создания заказа

Находит книгу для каждого идентификатора, используя LINQ-метод Contains, чтобы найти все ключи

Включает любые дополнительные акции, которые необходимы BizLogic для определения цены

Возвращает результат в виде словаря, чтобы BizLogic было проще их искать

Этот метод добавляет новый заказ в коллекцию Orders класса DbContext

Класс `PlaceOrderDbAccess` реализует интерфейс `IPlaceOrderDbAccess`, с помощью которого класс `PlaceOrderAction` получает доступ к этому классу. Помимо помощи с внедрением зависимостей, которая описана в главе 5, использование интерфейса позволяет заменить класс `PlaceOrderDbAccess` тестовой версией, – также называемой *заглушкой* или *имитацией*, – при модульном тестировании класса `PlaceOrderAction`. В разделе 17.7 эта тема рассматривается более подробно.

4.4.5 Правило 5: бизнес-логика не должна вызывать метод EF Core, SaveChanges

Последнее правило гласит, что бизнес-логика не должна вызывать метод EF Core `SaveChanges`, который обновляет базу данных напрямую. На то есть несколько причин:

- считается, что сервисный слой является основной точкой доступа к базе данных: он управляет тем, что записывается в базу данных;
- сервисный слой вызывает метод `SaveChanges` только в том случае, если бизнес-логика не возвращает ошибок.

Чтобы помочь вам управлять своей бизнес-логикой, я создал серию простых классов, которые использую для выполнения любой бизнес-логики. Я называю их `BizRunner`. Это обобщенные классы, способные вызывать бизнес-логику с разными типами ввода и вывода. Различные варианты `BizRunner` могут обрабатывать различные комбинации ввода/вывода и асинхронные методы (об использовании паттерна async/await с EF Core рассказывается в главе 5), а также некоторые методы с дополнительными функциями, такие как `PlaceOrderAction` (раздел 4.7.3).

Каждый `BizRunner` определяет обобщенный интерфейс, который должна реализовать бизнес-логика. Ваш класс из проекта `BizLogic` выполняет действие, которое ожидает один входной параметр типа `PlaceOrderInDto` и возвращает объект типа `Order`. Следовательно, класс `PlaceOrderAction` реализует интерфейс, как показано в следующем листинге, но со своими типами ввода и вывода (`IBizAction<PlaceOrderInDto,Order>`).

Листинг 4.4 Интерфейс, позволяющий `BizRunner` выполнять бизнес-логику

```
public interface IBizAction<in TIn, out TOut>
{
    IImmutableList<ValidationResult>
        Errors { get; }
    bool HasErrors { get; }
    TOut Action(TIn dto);
}
```

BizAction использует TIn и TOut для определения входных и выходных параметров метода Action

Возвращает информацию об ошибке из бизнес-логики

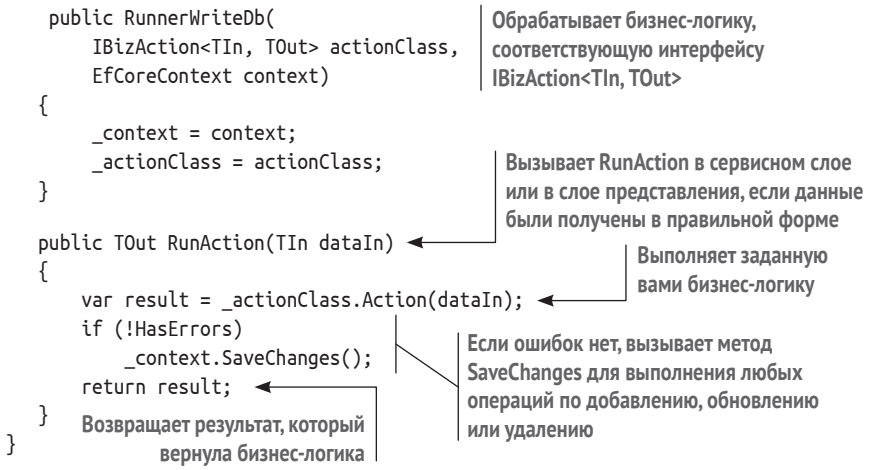
Действие, вызываемое BizRunner

Когда у вас есть класс бизнес-логики, реализующий этот интерфейс, `BizRunner` знает, как запустить данный код. Сам `BizRunner` небольшой, как показано в следующем листинге, из которого видно, что он называется `RunnerWriteDb<TIn, TOut>`. Этот вариант `BizRunner` предназначен для работы с бизнес-логикой, которая получает данные на вход, возвращает результат и производит запись в базу данных.

Листинг 4.5 `BizRunner`, запускающий бизнес-логику, возвращая результат или ошибки

```
public class RunnerWriteDb<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors => _actionClass.Errors;
    public bool HasErrors => _actionClass.HasErrors;
```

Информация об ошибках в бизнес-логике возвращается пользователю BizRunner



Паттерн BizRunner скрывает бизнес-логику и представляет обобщенный интерфейс / API, который могут использовать другие классы. Вызывающему коду `BizRunner` не нужно беспокоиться о EF Core, потому что все вызовы EF Core находятся в коде `BizDbAccess` или в `BizRunner`. Этот факт сам по себе является достаточной причиной для использования паттерна BizRunner, но, как вы увидите позже, он позволяет создавать другие формы `BizRunner` с дополнительной функциональностью.

ПРИМЕЧАНИЕ Вы можете посмотреть созданную мной библиотеку с открытым исходным кодом, которая называется `EfCore.GenericBizRunner`. Она предоставляет ту же функциональность, что и `BizRunner`, но в виде библиотеки и использует обобщенные классы, которые запускают бизнес-логику, не требуя написания дополнительного кода. Посетите страницу http://mng.bz/vz7J для получения дополнительной информации.

Один из важных моментов, связанных с `BizRunner`, состоит в том, что это должен быть единственный метод, которому разрешено вызывать метод `SaveChanges` в течение жизненного цикла DbContext приложения. Почему? Бизнес-логика не думает о базе данных, поэтому для нее вполне нормально добавлять или обновлять класс сущности в любое время, а ошибка может быть обнаружена позже. Чтобы изменения, внесенные до того, как была обнаружена ошибка, не записывались в базу данных, вы полагаетесь на то, что метод `SaveChanges` не будет вызываться бизнес-логикой в течение жизненного цикла DbContext приложения.

В ASP.NET-приложении управлять жизненным циклом DbContext довольно просто, поскольку для каждого HTTP-запроса создается новый экземпляр DbContext. В приложениях с длительным временем выполнения такая ситуация является проблемой. Раньше я избегал этого, заставляя BizRunner создавать новый скрытый экземпляр

DbContext приложения, чтобы быть уверенным в том, что никакой другой код не будет вызывать метод `SaveChanges` для этого экземпляра DbContext.

4.4.6 Собираем все вместе: вызов бизнес-логики для обработки заказов

Теперь, когда мы изучили все части этого сложного паттерна бизнес-логики, можно посмотреть, как вызвать этот код. В листинге 4.6 показан класс `PlaceOrderService` из сервисного слоя, вызывающий BizRunner для выполнения `PlaceOrderAction`, который выполняет обработку заказа.

> **ПРИМЕЧАНИЕ** Я использую HTTP-файл cookie, чтобы сохранить выбранные пользователем книги, которые он хочет купить. Я называю этот файл *cookie-корзиной*. Это работает потому, что HTTP-файл cookie может хранить небольшой объем данных на компьютере пользователя. Я использую cookie ASP.NET Core для доступа к cookie-корзине пользователя. Для получения дополнительной информации см. http://mng.bz/4ZNa.

Если бизнес-логика выполнилась успешно, то код очищает cookie-корзину и возвращает первичный ключ класса сущности `Order`, чтобы можно было показать страницу подтверждения пользователю. Если заказ не был успешным, cookie-корзина не очищается, и снова отображается страница оформления заказа с сообщениями об ошибках, чтобы пользователь мог исправить все проблемы и повторить попытку.

Листинг 4.6 Класс `PlaceOrderService`, вызывающий бизнес-логику

```
public class PlaceOrderService
{
    private readonly BasketCookie _basketCookie;

    private readonly
        RunnerWriteDb<PlaceOrderInDto, Order> _runner;
    public IImmutableList<ValidationResult>
        Errors => _runner.Errors;

    public PlaceOrderService(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
        _basketCookie = new BasketCookie(
            cookiesIn, cookiesOut);
```

Этот класс обрабатывает cookie-корзину, которая содержит выбранные пользователем книги

Определяет входной, PlaceOrderInDto, и выходной, Order, параметры этой бизнес-логики

Содержит все ошибки, отправленные из бизнес-логики

Конструктор принимает входящие и исходящие данные cookie, а также DbContext приложения

Создает BasketCookie, используя входящие и исходящие данные cookie из ASP.NET Core

```
            _runner =
                new RunnerWriteDb<PlaceOrderInDto, Order>(
                    new PlaceOrderAction(
                        new PlaceOrderDbAccess(context)),
                    context);
        }

        public int PlaceOrder(bool acceptTAndCs)
        {
            var checkoutService = new CheckoutCookieService(
                _basketCookie.GetValue());

            var order = _runner.RunAction(
                new PlaceOrderInDto(acceptTAndCs,
                checkoutService.UserId,
                checkoutService.LineItems));

            if (_runner.HasErrors) return 0;

            checkoutService.ClearAllLineItems();
            _basketCookie.AddOrUpdateCookie(
                checkoutService.EncodeForCookie());

            return order.OrderId;
        }
    }
```

Создает BizRunner с бизнес-логикой, которая должна быть запущена

Этот метод вызывается, когда пользователь нажимает кнопку «Купить»

Запускает бизнес-логику с необходимыми данными из cookie-корзины

CheckoutCookieService – это класс, который зашифровывает и расшифровывает данные корзины

Если у бизнес-логики есть ошибки, они немедленно возвращаются. Cookie-корзина не очищается

Заказ был размещен успешно, поэтому cookie-корзина очищается

Возвращает OrderId, что позволяет ASP.NET подтвердить детали заказа

Помимо выполнения бизнес-логики, этот класс действует как паттерн *Адаптер*; он преобразует данные из cookie-корзины в форму, которую принимает бизнес-логика, и при успешном завершении извлекает первичный ключ класса сущности `Order`, `OrderId`, чтобы отправить его обратно в слой представления ASP.NET Core.

Такая роль паттерна «Адаптер» типична для кода, вызывающего бизнес-логику, поскольку между форматами данных слоя представления и бизнес-логики часто возникает несоответствие. Оно может быть небольшим, как в этом примере, но вам, скорее всего, потребуется адаптировать все, кроме простейших вызовов бизнес-логики. Поэтому в моей более сложной библиотеке EfCore.GenericBizRunner есть готовая реализация паттерна «Адаптер».

4.4.7 Размещение заказа в приложении Book App

Теперь, когда мы рассмотрели бизнес-логику для обработки заказа, `BizRunner` и `PlaceOrderService`, который выполняет бизнес-логику, посмотрим, как использовать эту логику в контексте приложения Book App. Этот процесс показан на рис. 4.4: от нажатия пользователем кнопки «Купить» до выполнения бизнес-логики и возврата результата. Я не буду здесь вдаваться в подробности кода представления, так как эта глава посвящена использованию EF Core в бизнес-логике, но кое-что будет рассмотрено в главе 5, посвященной использованию EF Core в приложениях ASP.NET Core.

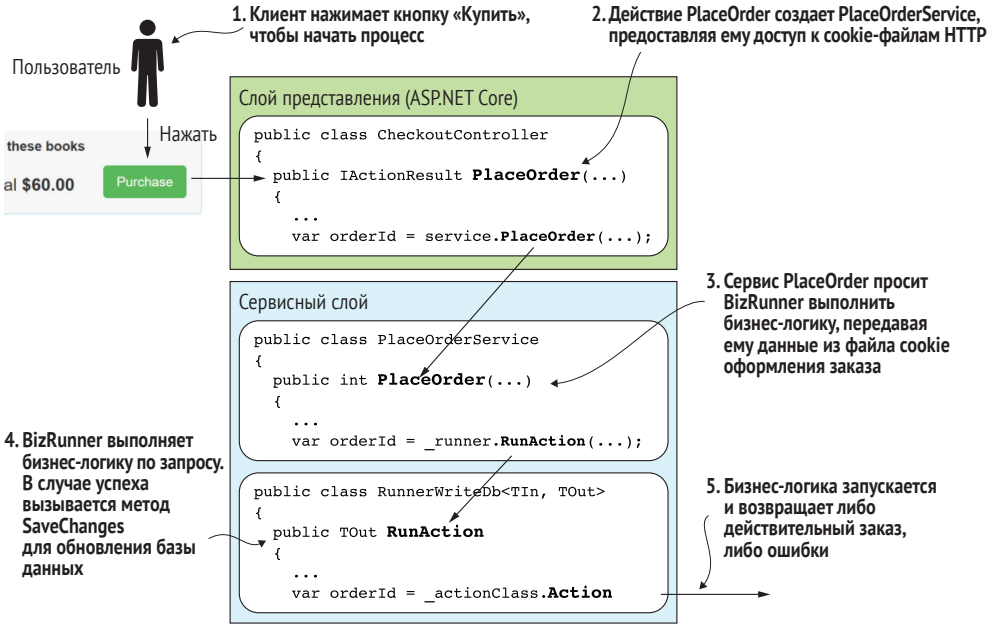


Рис. 4.4 Последовательность шагов от нажатия пользователем кнопки «Купить» до сервисного слоя, где `BizRunner` выполняет бизнес-логику для обработки заказа

При нажатии кнопки «Купить» на рис. 4.4 в `CheckoutController` выполняется действие ASP.NET Core `PlaceOrder`. Это действие создает класс `PlaceOrderService` из сервисного слоя, который содержит большую часть логики паттерна «Адаптер». Вызывающий код предоставляет этому классу доступ к cookie-файлам для чтения и записи, в то время как данные хранятся в HTTP-cookie на устройстве пользователя.

Вы видели класс `PlaceOrderService` в листинге 4.6. Его метод `PlaceOrder` извлекает данные заказа из HTTP-cookie и создает объект DTO в форме, которая требуется бизнес-логике. Затем он вызывает обобщенный `BizRunner` для запуска бизнес-логики, которую он должен выполнить. Когда `BizRunner` возвращается из бизнес-логики, возможны два пути:

- *заказ был размещен успешно* (*ошибок нет*). В этом случае метод `PlaceOrder` очищает cookie-корзину и возвращает `OrderId` размещенного заказа, поэтому код ASP.NET Core может отобразить страницу подтверждения с описанием заказа;
- *заказ был неудачным* (*есть ошибки*). В этом случае метод `PlaceOrder` незамедлительно возвращается к коду ASP.NET Core, который обнаруживает ошибки, повторно отображает страницу оформления заказа и добавляет сообщения об ошибках, чтобы пользователь мог их исправить и повторить попытку.

ПРИМЕЧАНИЕ Можно опробовать процесс оформления заказа, скачав код приложения Book App и запустив его локально, чтобы увидеть результаты. Чтобы опробовать ошибки, не ставьте галочку напротив поля «Пользовательское соглашение».

4.4.8 Плюсы и минусы паттерна сложной бизнес-логики

Я использовал этот паттерн в течение многих лет и считаю, что в целом это отличный подход, но он требует большого количества кода. Я хочу сказать, что нужно писать дополнительный структурный код для его реализации. Поэтому использую его только для сложной бизнес-логики. В следующих разделах подробно рассматриваются преимущества и недостатки этого паттерна.

ПРЕИМУЩЕСТВА ДАННОГО ПАТТЕРНА

Этот паттерн следует DDD-подходу, который пользуется большим уважением и широко распространен. Он сохраняет бизнес-логику «чистой» в том плане, что она не знает о базе данных, скрытой с помощью методов `BizDbAccess`, которые предоставляют репозиторий для бизнес-логики. Кроме того, класс `BizDbAccess` позволяет тестировать бизнес-логику без использования базы данных, поскольку модульные тесты могут предоставить класс для замены (известный как заглушка или имитация), который может предоставить тестовые данные по мере необходимости.

НЕДОСТАТКИ ДАННОГО ПАТТЕРНА

Ключевой недостаток состоит в том, что нужно писать больше кода, чтобы отделить бизнес-логику от доступа к базе данных, что требует больше времени и усилий. Если бизнес-логика проста или большая часть кода работает с базой данных, усилия по созданию отдельного класса для обработки доступа к базе данных не имеют смысла.

4.5 Пример простой бизнес-логики: ChangePriceOfferService

В моем примере простой бизнес-логики мы создадим бизнес-логику для добавления или удаления акционной цены на книгу. В этом примере есть бизнес-правила, но, как вы увидите, они связаны с большим количеством обращений к базе данных. Вот эти правила:

- если у `Book` есть `PriceOffer`, нужно удалить текущий `PriceOffer` (убрать акционную цену);
- если у `Book` нет `PriceOffer`, мы добавляем новую акционную цену;
- при добавлении акционной цены значение `PromotionalText` не должно быть равно null или быть пустым.

Как вы увидите в разделе 4.5.2, этот код представляет собой смесь бизнес-правил и обращений к базе данных, которую я определяю как простой тип бизнес-логики.

4.5.1 Мой подход к проектированию простой бизнес-логики

В случае с простой бизнес-логикой мне нужна минимальная по сложности структура проекта, потому что я считаю, что бизнес-логика достаточно проста и/или настолько взаимосвязана с доступом к базе данных, что ее не нужно изолировать. В результате пять правил, изложенных в разделе 4.3.1, не используются, что ускоряет создание кода. Обратная сторона состоит в том, что бизнес-логика смешана с другим кодом, а это может затруднить понимание бизнес-логики и усложнить модульное тестирование – компромиссы, с которыми вам придется иметь дело для более быстрой разработки.

Как правило, я размещаю простую бизнес-логику в сервисном слое, а не в слое BizLogic, потому что ей необходим доступ к DbContext приложения, а слой BizLogic не разрешает этого. Обычно я помещаю свою простую бизнес-логику в классы CRUD, которые работают над той же функциональностью. В `ChangePriceOfferService` я помещаю класс `ChangePriceOfferService` в папку `AdminServices` наряду с другими CRUD-сервисами.

4.5.2 Пишем код класса ChangePriceOfferService

Класс `ChangePriceOfferService` содержит два метода: метод `GetOriginal`, представляющий собой простую CRUD-команду для загрузки `PriceOffer`, и метод `AddRemovePriceOffer`, который отвечает за создание или удаление класса `PriceOffer` для книги. Второй метод содержит бизнес-логику и показан в следующем листинге.

Листинг 4.7 Метод `AddRemovePriceOffer` в классе `ChangePriceOfferService`

Этот метод удаляет PriceOffer, если он присутствует; в противном случае добавляется новый PriceOffer

```
public ValidationResult AddRemovePriceOffer(PriceOffer promotion)
{
    var book = _context.Books
        .Include(r => r.Promotion)
        .Single(k => k.BookId
                     == promotion.BookId);

    if (book.Promotion != null)
    {
        _context.Remove(book.promotion);
        _context.SaveChanges();
        return null;
    }
```

Загружает книгу с существующей акцией

Если у книги есть промоакция, удаляет эту акцию

Удаляет запись PriceOffer, связанную с выбранной книгой

Возвращает null, а это означает, что метод успешно завершил работу

```
if (string.IsNullOrEmpty(promotion.PromotionalText))
{
    return new ValidationResult(
        "This field cannot be empty",
        new []{ nameof(PriceOffer.PromotionalText)});
}
book.Promotion = promotion;
 _context.SaveChanges();
 return null;
}
```

Проверка валидности. PromotionalText должен содержать некий текст

Возвращает сообщение об ошибке с именем свойства, содержащего неправильное значение

Назначает новый PriceOffer выбранной книге

Метод SaveChanges обновляет базу данных

Добавление новой промоакции прошло успешно, поэтому метод возвращает значение null

4.5.3 Плюсы и минусы этого паттерна бизнес-логики

Мы написали бизнес-логику, реализованную иначе, чем более сложная бизнес-логика для обработки заказа, которую я описал как простую бизнес-логику. Основные различия между ними заключаются в следующем:

- простая бизнес-логика не соответствовала правилам DDD из раздела 4.3.1. В частности, она не изолировала доступ к базе данных от бизнес-логики;
- простая бизнес-логика была размещена на сервисном слое (а не в слое BizLogic) вместе с CRUD-сервисами, относящимися к корзине.

У этого паттерна есть свои достоинства и недостатки.

ПРЕИМУЩЕСТВА ДАННОГО ПАТТЕРНА

Данный паттерн практически не имеет заданной структуры, поэтому, чтобы достигнуть нужной бизнес-цели, можно писать код самым простым способом. Обычно код будет короче, чем при использовании паттерна сложной бизнес-логики, в котором есть дополнительные классы, чтобы изолировать бизнес-логику от базы данных.

Бизнес-логика также является самодостаточной, и весь код находится в одном месте. В отличие от примера со сложной бизнес-логикой, эта бизнес-логика обрабатывает все. Например, для ее выполнения не требуется BizRunner, потому что код сам вызывает метод `SaveChanges`, что упрощает изменение, перемещение и тестирование, поскольку он не полагается ни на что другое.

Кроме того, поместив классы бизнес-логики в сервисный слой, я могу сгруппировать эти сервисы простой бизнес-логики в той же папке, что и CRUD-сервисы, связанные с данной функцией. В результате я могу быстро найти весь базовый код функции, потому что код сложной бизнес-логики находится в другом проекте.

НЕДОСТАТКИ ДАННОГО ПАТТЕРНА

У вас нет подхода, основанного на концепции DDD, поэтому ответственность за рациональное проектирование бизнес-логики лежит

на вас. Ваш опыт поможет вам выбрать лучший паттерн и написать правильный код. Главное здесь – простота. Если код прост для понимания, значит, вы все правильно сделали; в противном случае код будет слишком сложным и должен следовать паттерну сложной бизнес-логики.

4.6 Пример валидации: добавление отзыва в книгу с проверкой

Последний пример – это обновление для примера с CRUD из главы 3. Там мы добавили `Review` в `Book`. Но в примере отсутствовали некоторые важные бизнес-правила:

- значение свойства `NumStars` должно быть от 0 до 5;
- в свойстве `Comment` должен быть текст.

В этом разделе мы обновим CRUD-код, добавив валидацию. В следующем листинге показан улучшенный метод `AddReviewWithChecks`, но основное внимание уделяется части проверки.

Листинг 4.8 Улучшенный CRUD-код с добавлением валидации

Этот метод добавляет отзыв в книгу с валидацией данных

Создает экземпляр класса состояния для хранения любых ошибок

```
public IStatusGeneric AddReviewWithChecks(Review review)
{
    var status = new StatusGenericHandler();
    if (review.NumStars < 0 || review.NumStars > 5)
        status.AddError("This must be between 0 and 5.",
            nameof(Review.NumStars));
    if (string.IsNullOrWhiteSpace(review.Comment))
        status.AddError("Please provide a comment with your review.",
            nameof(Review.Comment));
    if (!status.IsValid)
        return status;

    var book = _context.Books
        .Include(r => r.Reviews)
        .Single(k => k.BookId
                     == review.BookId);
    book.Reviews.Add(review);
    _context.SaveChanges();
    return status;
    }
```

Добавляет ошибку в экземпляр состояния, если рейтинг находится вне нужного диапазона

Вторая проверка гарантирует, что пользователь оставил какой-то комментарий

Если есть какие-либо ошибки, метод немедленно завершается, возвращая экземпляр класса с ошибками

CRUD-код, добавляющий отзыв в книгу

Возвращает экземпляр состояния, сообщающего о наличии или отсутствии ошибок

ПРИМЕЧАНИЕ Используемые в листинге 4.8 интерфейс `IStatusGeneric` и класс `StatusGenericHandler` взяты из пакета NuGet, GenericServices.StatusGeneric. Эта библиотека предоставляет

простой, но исчерпывающий способ вернуть состояние, соответствующее подходу к валидации .NET Core. Сопутствующий пакет NuGet, `EfCore.GenericServices.AspNetCore`, предоставляет способы преобразования состояния `IStatusGeneric` в страницы на базе Razor с `ModelState` из ASP.NET Core или коды состояния HTTP для контроллеров веб-API.

Это CRUD-метод с добавлением валидации, что типично для данного типа бизнес-логики. В этом случае мы использовали код с `if-then` для проверки свойства, но вместо этого можно было бы использовать `DataAnnotations`. Как я сказал ранее, данный тип проверки обычно выполняется на стороне клиента, но дублирование проверки конфиденциальных данных в коде серверной части может сделать приложение более надежным. Позже, в разделе 4.7.1, я покажу вам, как проверять данные перед их записью в базу данных, что даст вам еще один вариант.

4.6.1 Плюсы и минусы этого паттерна бизнес-логики

Бизнес-логика валидации – это CRUD-сервисы, которые вы видели в главе 3, с добавлением валидации. Поэтому я помещаю классы бизнес-логики валидации в сервисный слой наряду с другими сервисами CRUD.

ПРЕИМУЩЕСТВА ДАННОГО ПАТТЕРНА

Вы уже знаете о CRUD-сервисах из главы 3, поэтому вам не нужно изучать еще один паттерн – только добавьте проверки и верните состояние. Однако, как и многие другие, я считаю эти классы бизнес-логики валидации такими же, как CRUD-сервисы с дополнительными проверками.

НЕДОСТАТКИ ДАННОГО ПАТТЕРНА

Единственный минус состоит в том, что нужно что-то делать с состоянием, которое возвращает паттерн, например повторно отображает форму ввода с сообщением об ошибке. Но это обратная сторона предоставления дополнительной проверки, а не проектирования бизнес-логики валидации.

4.7 Добавление дополнительных функций в обработку вашей бизнес-логики

Этот паттерн для обработки бизнес-логики упрощает добавление дополнительных функций. В данном разделе мы добавим две функции:

- проверка класса сущности для метода `SaveChanges`;

- транзакции, которые объединяют код бизнес-логики в одну логическую атомарную операцию.

Эти функции используют команды EF Core, которые не ограничиваются бизнес-логикой. Обе функции могут применяться в других областях, поэтому, возможно, вам придется помнить о них, когда вы будете работать над своим приложением.

4.7.1 Валидация данных, которые вы записываете в базу

Я уже говорил о проверке данных до того, как они попадут в базу, а в этом разделе показано, как добавить проверку при записи в базу данных. .NET содержит целую экосистему для валидации данных, проверки значения свойства на соответствие определенным правилам (например, находится ли целое число в диапазоне от 1 до 10 или длина строки не более 20 символов). Эта экосистема используется во многих frontend-системах Microsoft.

> **EF6** Если вы ищете изменения, касающиеся EF6.x, читайте следующий абзац. Метод `SaveChanges` не проверяет данные перед их записью в базу, но в этом разделе показано, как добавить эту проверку.

В предыдущей версии EF (EF6.x) добавляемые или обновляемые данные проверялись по умолчанию перед записью в базу данных. В EF Core, который разработан, чтобы быть легковеснее и быстрее, валидация не выполняется при добавлении данных или обновлении базы данных. Идея состоит в том, что проверка часто выполняется в клиентской части, так зачем повторять ее?

Как вы уже видели, бизнес-логика содержит большое количество кода валидации, и часто бывает полезно поместить этот код в классы сущностей в качестве валидации, особенно если ошибка связана с определенным свойством в классе сущности. Этот пример – еще один случай разбиения сложного набора правил на несколько частей.

В листинге 4.9 тест для проверки того, продается ли книга, переносится из бизнес-логики в код валидации. Кроме того, сюда добавлены две новые проверки, чтобы продемонстрировать вам различные формы, которые могут принимать результаты этих проверок, делая данный пример более всеобъемлющим.

На рис. 4.5 показан класс сущности `LineItem` с двумя способами задания правил валидации. Первый способ – это атрибут `[Range(min, max)]`, известный как Data Annotations (см. раздел 7.4). Он добавляется к свойству `LineNum`. Второй способ – это интерфейс `IValidatableObject`, требующий, чтобы вы добавили метод `IValidatableObject.Validate`, в котором можно написать собственные правила проверки и возвращать ошибки, если эти правила нарушаются.

Листинг 4.9 Правила проверки, применяемые к классу сущности `LineItem`

```
public class LineItem : IValidatableObject
{
    public int LineItemId { get; set; }

    [Range(1,5, ErrorMessage =
        "This order is over the limit of 5 books.")]
    public byte LineNum { get; set; }

    public short NumBooks { get; set; }

    public decimal BookPrice { get; set; }

    // Связи;

    public int OrderId { get; set; }
    public int BookId { get; set; }

    public Book ChosenBook { get; set; }

    IEnumerable<ValidationResult> IValidatableObject.Validate
        (ValidationContext validationContext)
    {
        var currContext =
            validationContext.GetService(typeof(DbContext));

        if (ChosenBook.Price < 0)
            yield return new ValidationResult(
$"Sorry, the book '{ChosenBook.Title}' is not for sale.");

        if (NumBooks > 100)
            yield return new ValidationResult(
     "If you want to order a 100 or more books"+
" please phone us on 01234-5678-90",
                new[] { nameof(NumBooks) });
    }
}
```

Интерфейс IValidatableObject добавляет метод IValidatableObject.Validate

Добавляет сообщение об ошибке, если свойство LineNum выходит за пределы допустимого диапазона

Разрешает доступ к текущему DbContext, если необходимо получить дополнительную информацию

Интерфейс IValidatableObject требует создания этого метода

Перемещает проверку цены из бизнес-логики в этот метод

Правило дополнительной проверки: для заказа более 100 книг необходимо позвонить, чтобы оформить заказ

Возвращает имя свойства с ошибкой, чтобы предоставить более подходящее сообщение об ошибке

Должен отметить, что в методе `IValidatableObject.Validate` вы получаете доступ к свойству за пределами класса `LineItem`: заголовку выбранной книги (`ChosenBook`). `ChosenBook` – это навигационное свойство, и при вызове метода `DetectChanges` ссылочная фиксация (см. рис. 1.10, этап 3) гарантирует, что свойство `ChosenBook` не имеет значения NULL.

В результате код валидации из листинга 4.9 может получить доступ к навигационным свойствам, которых у бизнес-логики может не быть.

ПРИМЕЧАНИЕ Помимо использования обширного списка встроенных атрибутов проверки, можно создавать собственные атрибуты, наследуя от класса `ValidationAttribute`. Посети-

те страницу http://mng.bz/9cec для получения дополнительной информации о доступных стандартных атрибутах проверки и о том, как использовать класс `ValidationAttribute`.

После добавления кода правила проверки в класс сущности `LineItem` необходимо добавить этап проверки в метод `SaveChanges`, который называется `SaveChangesWithValidation`. Хотя очевидное место для его размещения – это DbContext приложения, мы создадим метод расширения. Он позволит использовать `SaveChangesWithValidation` в любом DbContext, а это означает, что вы можете скопировать этот класс и использовать его в своем приложении.

В следующем листинге показан этот метод расширения `SaveChangesWithValidation`, а в листинге 4.11 представлен закрытый метод `ExecuteValidation`, который вызывает `SaveChangesWithValidation` для обработки проверки.

Листинг 4.10 Метод `SaveChangesWithValidation`, добавленный для DbContext приложения

```
public static ImmutableList<ValidationResult>
    SaveChangesWithValidation(this DbContext context)
{
    var result = context.ExecuteValidation();

    if (result.Any()) return result;

    context.SaveChanges();

    return result;
}
```

SaveChangesWithValidation возвращает список объектов типа ValidationResult

SaveChangesWithValidation – это метод расширения, который принимает DbContext в качестве входных данных

Возвращает пустой список, чтобы показать, что ошибок нет

ExecuteValidation используется в SaveChangesWithChecking/ SaveChangesWithCheckingAsync

Если есть ошибки, немедленно выходим и не вызываем метод SaveChange

Ошибок нет, поэтому вызываем метод SaveChanges

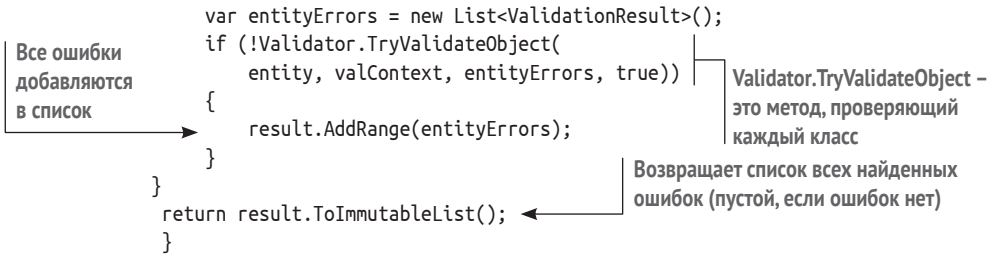
Листинг 4.11 Метод `SaveChangesWithValidation` вызывает метод `ExecuteValidation`

```
private static ImmutableList<ValidationResult>
    ExecuteValidation(this DbContext context)
{
    var result = new List<ValidationResult>();
    foreach (var entry in
        context.ChangeTracker.Entries()
            .Where(e =>
                (e.State == EntityState.Added) ||
                (e.State == EntityState.Modified)))
{
        var entity = entry.Entity;
        var valProvider = new
            ValidationDbContextServiceProvider(context);
        var valContext = new
            ValidationContext(entity, valProvider, null);
```

Использует ChangeTracker, чтобы получить доступ ко всем классам сущностей, которые он отслеживает

Фильтрует сущности, которые будут добавлены или обновлены в базе данных

Реализует интерфейс IServiceProvider и передает DbContext методу Validate



Основной код проверки находится в методе `ExecuteValidation`, потому что он должен использоваться как в синхронных, так и в асинхронных версиях метода `SaveChangesWithValidation`. Вызов `context.ChangeTracker.Entries` вызывает `DetectChanges` DbContext, чтобы убедиться, что все внесенные вами изменения найдены до запуска валидации. После этого мы просматриваем все добавленные или измененные (обновленные) сущности и проверяем их.

Здесь есть фрагмент кода, который я хочу выделить, – это класс `ValidationDbContextServiceProvider`. Он реализует интерфейс `IServiceProvider`. Этот класс используется при создании `ValidationContext`, поэтому он доступен во всех классах сущностей, у которых есть интерфейс `IValidatableObject`. Это позволяет методу `Validate` при необходимости обращаться к DbContext текущего приложения. Наличие доступа к текущему DbContext позволяет создавать более подходящие сообщения об ошибках, получая дополнительную информацию из базы данных.

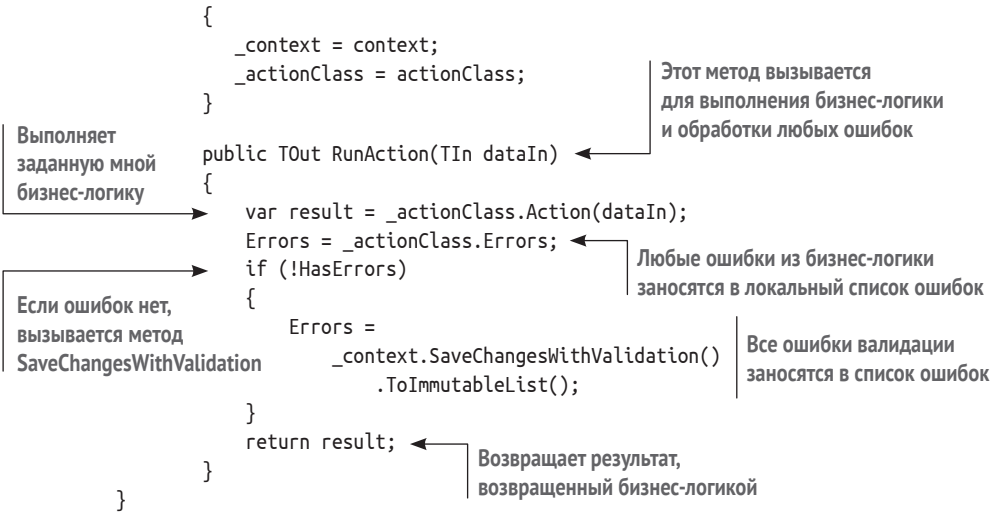
Мы проектируем метод `SaveChangesWithValidation` таким образом, чтобы он возвращал ошибки, а не генерировал исключение. Это делается для того, чтобы соответствовать бизнес-логике, которая возвращает ошибки в виде списка, а не исключения. Можно создать новый вариант `BizRunner`, `RunnerWriteDbWithValidation`, использующий метод `SaveChangesWithValidation` вместо обычного метода `SaveChanges` и возвращающий ошибки бизнес-логики или любые ошибки проверки, обнаруженные при записи в базу данных. В следующем листинге показан класс `RunnerWeriteDbWithValidation`.

Листинг 4.12 Вариант `BizRunner`, класс `RunnerWriteDbWithValidation`

```
public class RunnerWriteDbWithValidation<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerWriteDbWithValidation(
        IBizAction<TIn, TOut> actionClass,
        EfCoreContext context)
```

Для этой версии необходимы собственные свойства Errors/HasErrors, поскольку ошибки идут из двух источников

Обрабатывает бизнес-логику, соответствующую интерфейсу IBizAction <TIn, TOut>



Приятная особенность этого нового варианта паттерна `BizRunner` заключается в том, что у него точно такой же интерфейс, что и у исходного `BizRunner`, где нет валидации. Можно заменить `RunnerWriteDbWithValidation<TIn, TOut>` для исходного `BizRunner` без необходимости изменять бизнес-логику или способ, которым вызывающий метод выполняет `BizRunner`.

В разделе 4.7.2 мы создадим еще один вариант `BizRunner`, который может запускать несколько классов бизнес-логики таким образом, чтобы они выглядели как один метод. Подобное возможно благодаря паттерну бизнес-логики, описанному в начале данной главы.

4.7.2 Использование транзакций для объединения кода бизнес-логики в одну логическую атомарную операцию

Как я сказал ранее, бизнес-логика может быть сложной. Когда дело доходит до проектирования и реализации крупной или сложной бизнес-логики, есть три варианта:

- *вариант 1* – написать один большой метод, который делает все;
- *вариант 2* – написать несколько мелких методов с одним главным методом, который выполнит их последовательно;
- *вариант 3* – написать несколько мелких методов, каждый из которых обновляет базу данных, но объединить их в одну Единицу работы (*Unit Of Work*) (см. врезку в разделе 3.2.2).

Вариант 1 – обычно плохая идея, потому что метод будет очень сложным для понимания и рефакторинга. Кроме того, у него могут быть проблемы, если части бизнес-логики используются где-то еще, потому что вы можете нарушить принцип разработки программного обеспечения DRY (Don't repeat yourself – Не повторяйся).

Вариант 2 может сработать, но у него могут быть проблемы, если поздние этапы бизнес-логики полагаются на элементы базы данных, записанных на ранних этапах, а это может нарушить правило атомарности, упомянутое в главе 1: когда в базу данных вносится несколько изменений, они либо все успешны, либо все терпят неудачу.

Остается вариант 3, который возможен благодаря функциональности EF Core (и большинства реляционных баз данных), называемой *транзакциями*. В разделе 3.2.2 на врезке «Почему следует вызывать метод `SaveChanges` только один раз в конце ваших изменений» вы познакомились с паттерном Единица работы. Там было показано, как метод `SaveChanges` сохраняет все изменения внутри транзакции, чтобы убедиться, что они были сохранены или, если база данных отклонила какую-то часть изменения, что никакие изменения не были сохранены в базе данных.

В нашем случае нужно распределить паттерн Единица работы по нескольким более мелким методам; назовем их `Biz1`, `Biz2` и `Biz3`. Не нужно менять методы `Biz`; они по-прежнему думают, что работают сами по себе, и будут ожидать вызова метода `SaveChanges` по завершении каждого метода `Biz`. Но при создании транзакции все три метода `Biz` с вызовом `SaveChanges` будут работать как одна Единица работы. В результате отказ или ошибка базы данных в `Biz3` отменит любые изменения в базе данных, сделанные `Biz1`, `Biz2` и `Biz3`.

Эта отмена работает, потому что использование EF Core для создания явной транзакции реляционной базы данных дает два эффекта:

- любая запись в базу данных скрыта от других пользователей базы данных, пока вы не вызовете метод транзакции `Commit`;
- если вы решите, что не хотите, чтобы в базу данных были записаны изменения (скажем, потому что в бизнес-логике есть ошибка), то можно отменить все изменения, сделанные в транзакции, вызвав команду транзакции `RollBack`.

На рис. 4.5 показаны три отдельные части бизнес-логики, каждая из которых ожидает вызова метода `SaveChanges` для обновления базы данных, но выполняемые классом, который называется *транзакционный* `BizRunner`. После выполнения каждой части бизнес-логики `BizRunner` вызывает метод `SaveChanges`, а это означает, что все, что записывает бизнес-логика, теперь доступно для последующих этапов бизнес-логики через локальную транзакцию.

На последнем этапе бизнес-логика, `Biz 3`, возвращает ошибки, заставляющие `BizRunner` вызвать команду `RollBack`, что приводит к удалению всех изменений, выполненных `Biz 1` и `Biz 2`.

В следующем листинге показан код нового транзакционного `BizRunner`, который запускает транзакцию в DbContext приложения перед вызовом любой бизнес-логики.

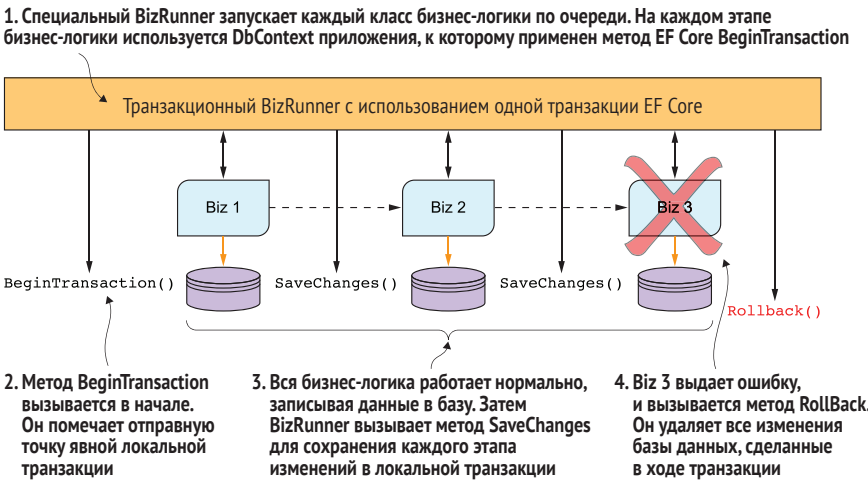


Рис. 4.5 Пример выполнения трех отдельных этапов бизнес-логики в рамках одной транзакции. Когда последний этап бизнес-логики возвращает ошибку, другие изменения в базе данных, сделанные на первых двух этапах бизнес-логики, откатываются

Листинг 4.13 RunnerTransact2WriteDb, последовательно выполняющий два этапа бизнес-логики

Три типа параметров – это входной класс, передаваемый из первой части бизнес-логики во вторую, и выходной

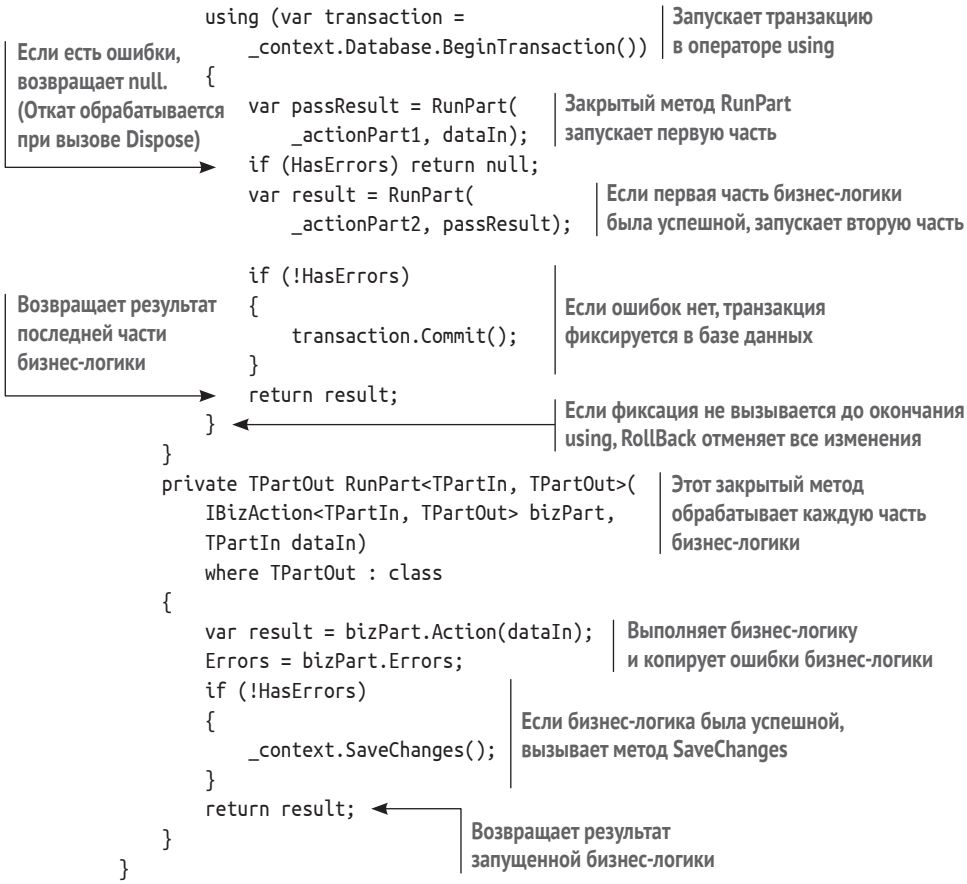
BizRunner может вернуть null, если есть ошибки, поэтому это должен быть класс

```
public class RunnerTransact2WriteDb<TIn, TPass, TOut>
    where TOut : class
{
    private readonly IBizAction<TIn, TPass>
        _actionPart1;
    private readonly IBizAction<TPass, TOut>
        _actionPart2;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerTransact2WriteDb(
        EfCoreContext context,
        IBizAction<TIn, TPass> actionPart1,
        IBizAction<TPass, TOut> actionPart2)
    {
        _context = context;
        _actionPart1 = actionPart1;
        _actionPart2 = actionPart2;
    }

    public TOut RunAction(TIn dataIn)
    {
```

Определяет обобщенный BizAction для двух частей бизнес-логики

Содержит любую информацию об ошибках, возвращаемую бизнес-логикой

Конструктор принимает классы бизнес-логики и DbContext приложения



В классе `RunnerTransact2WriteDb` вы по очереди выполняете каждую часть бизнес-логики, а в конце выполнения каждой части совершаете одно из следующих действий:

- *нет ошибок* – вы вызываете метод `SaveChanges`, чтобы сохранить в транзакцию изменения, выполненные бизнес-логикой. Это сохранение выполняется в рамках локальной транзакции, поэтому другие методы с доступом к базе данных пока не увидят эти изменения. Затем вы вызываете следующую часть бизнес-логики, если она есть;
- *есть ошибки* – вы копируете ошибки, обнаруженные бизнес-логикой, которая только что завершила работу, в список ошибок `BizRunner` и выходите из `BizRunner`. В этот момент код выходит за пределы оператора `using`, содержащего транзакцию, что приводит к удалению транзакции. Поскольку метод транзакции `Commit` не был вызван, удаление заставит транзакцию выполнить метод `RollBack`, который отбрасывает все изменения базы данных, находящиеся в транзакции. Эти изменения никогда не попадут в базу данных.

Если вы выполнили всю бизнес-логику без ошибок, то вызываете команду `Commit`. Она выполняет атомарное обновление базы данных, чтобы отразить все изменения, содержащиеся в локальной транзакции, в базу.

4.7.3 Использование класса RunnerTransact2WriteDb

Чтобы протестировать класс `RunnerTransact2WriteDb`, мы разделим код обработки заказов, который использовали ранее, на две части:

- `PlaceOrderPart1` – создает сущность `Order` без `LineItems`;
- `PlaceOrderPart2` – добавляет `LineItems` для каждой купленной книги в сущность `Order`, созданную классом `PlaceOrderPart1`.

`PlaceOrderPart1` и `PlaceOrderPart2` основаны на коде `PlaceOrderAction`, который вы уже видели, поэтому я не буду здесь повторять код бизнес-логики.

В листинге 4.14 показаны изменения в коде, необходимые для `PlaceOrderService` (показано в листинге 4.6), чтобы переключиться на использование `RunnerTransact2WriteDb` `BizRunner`.

В листинге основное внимание уделяется той части, которая создает и запускает два этапа, `Part1` и `Part2`. Оставшиеся прежними части кода опущены, чтобы можно было легко увидеть изменения.

Листинг 4.14 Класс `PlaceOrderServiceTransact`, показывающий изменения в коде

```
public class PlaceOrderServiceTransact
{
    //… Остальной код удален, поскольку
      он такой же, как и в листинге 4.5;

    public PlaceOrderServiceTransact(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
        _checkoutCookie = new CheckoutCookie(
            cookiesIn, cookiesOut);
        _runner = new RunnerTransact2WriteDb
            <PlaceOrderInDto, Part1ToPart2Dto, Order>(
            context,
            new PlaceOrderPart1(
                new PlaceOrderDbAccess(context)),
            new PlaceOrderPart2(
                new PlaceOrderDbAccess(context)));
    }

    public int PlaceOrder(bool tsAndCsAccepted)
    {
        //… Остальной код удален, поскольку он такой же, как и в листинге 4.6;
    }
}
```

Эта версия PlaceOrderService использует транзакции для выполнения двух классов бизнес-логики: PlaceOrderPart1 и PlaceOrderPart2

Этот BizRunner обрабатывает несколько частей бизнес-логики внутри транзакции

BizRunner нужны входные данные, класс, переданный из Part1 в Part2, и выходные данные.

BizRunner нужен DbContext приложения

Предоставляет экземпляр первой части бизнес-логики

Предоставляет экземпляр второй части бизнес-логики

Важно отметить, что бизнес-логика не знает, что она выполняется в транзакции. Поэтому можно использовать части бизнес-логики как по отдельности, так и в одной транзакции. Точно так же листинг 4.14 показывает, что нужно изменить только вызывающий код транзакционной бизнес-логики, которую я называю `BizRunner`. Использование транзакции позволяет с легкостью объединить несколько классов бизнес-логики в одну транзакцию без необходимости изменять какой-либо код бизнес-логики.

Преимущество использования транзакций, подобных этой, заключается в том, что вы можете повторно применять части своей бизнес-логики, заставляя эти несколько вызовов обращаться к приложению, особенно к его базе данных, как один вызов. Я использовал этот подход, когда мне нужно было создать, а затем сразу же обновить сложную, состоящую из нескольких частей сущность. Поскольку мне нужна была бизнес-логика Update для других случаев, я использовал транзакцию для вызова бизнес-логики Create, за которой следовала бизнес-логика Update, что сэкономило мне усилия на разработку, а мой код следовал принципу DRY.

Недостаток этого подхода состоит в том, что он усложняет доступ к базе данных, а это может несколько усложнить отладку, или использование транзакций базы данных может вызвать проблемы с производительностью. Также имейте в виду, что если вы используете опцию `EnableRetryOnFailure` (см. раздел 11.8) для повторного доступа к базе данных при ошибках, необходимо обработать возможные множественные вызовы вашей бизнес-логики.

Резюме

- Термин *бизнес-логика* описывает код, написанный для реализации реальных бизнес-правил. Код бизнес-логики может быть простым или сложным.
- В зависимости от сложности бизнес-логики необходимо выбрать подход, соблюдающий баланс между простотой решения и временем, необходимым для разработки и тестирования вашего решения.
- Изоляция части бизнес-логики, связанной с доступом к базе данных, в другом классе или проекте может упростить написание кода чистой бизнес-логики, но на разработку уйдет больше времени.
- Объединить всю бизнес-логику для какой-либо функциональности в один класс можно быстро и легко, но это может усложнить понимание и тестирование кода.
- Создание стандартизированного интерфейса для бизнес-логики значительно упрощает вызов и выполнение бизнес-логики для клиентской части.
- Иногда проще перенести часть логики проверки в классы сущностей и выполнять проверки, когда эти данные записываются в базу данных.

- Для сложной или повторно используемой бизнес-логики возможно проще использовать транзакцию базы данных, чтобы обеспечить последовательное выполнение частей бизнес-логики, но с точки зрения базы данных они будут выглядеть как одна атомарная единица.

Для читателей, знакомых с EF6.x:

- в отличие от EF6.x, метод `SaveChanges` в EF Core не проверяет данные, перед тем как они будут записаны в базу данных. Но можно легко реализовать метод, предоставляющий такую возможность в EF Core.

5 Использование EF Core в веб-приложениях ASP.NET Core

В этой главе рассматриваются следующие темы:

- использование EF Core в ASP.NET Core;
- использование внедрения зависимостей в ASP.NET Core;
- доступ к базе данных в MVC-действиях ASP.NET Core;
- использование миграций EF Core для обновления базы данных;
- применение async/await для улучшения масштабируемости.

В этой главе мы соберем все воедино, используя ASP.NET Core для создания реального веб-приложения. Применение ASP.NET Core затрагивает проблемы, выходящие за рамки EF Core, например внедрение зависимостей (рассматривается в разделе 5.4) и async/await (раздел 5.10). Но они необходимы, если вы собираетесь применять EF Core в приложениях такого типа.

Предполагается, что вы уже прочитали главы 2–4 и знаете о запросах и обновлении базы данных, а также имеете представление о бизнес-логике. В этой главе рассказывается, где разместить код доступа к базе данных и как вызвать его в реальном приложении. Кроме того, она охватывает конкретные вопросы использования EF Core в приложениях ASP.NET Core (включая Blazor Server). По этой причине здесь довольно много говорится об ASP.NET Core, но все внимание сосредоточено на правильном использовании EF Core в этих приложениях.

В заключение я приведу более общие сведения о различных способах получения экземпляра DbContext для таких случаев, как фоновые задачи.

5.1 Знакомство с ASP.NET Core

На сайте ASP.NET Core сказано, что «ASP.NET Core – это кросс-платформенный высокопроизводительный фреймворк с открытым исходным кодом для создания современных облачных приложений, подключенных к интернету» (http://mng.bz/QmOw). Хорошее определение, но в ASP.NET Core так много замечательных функций, что сложно выбрать, какие из них стоит прокомментировать.

ПРИМЕЧАНИЕ Я рекомендую книгу Эндрю Лока «*ASP.NET Core в действии*», где подробно описаны многие функции ASP.NET Core.

Я много лет использовал ASP.NET MVC5, предшественника ASP.NET Core, и думал, что это хороший фреймворк, хотя и немного медленный в плане производительности. Но для меня ASP.NET Core затмил ASP.NET MVC5. Он феноменально улучшает производительность и предоставляет новые способы отображения данных, такие как Razor Pages и Blazor.

СОВЕТ Когда я впервые попробовал ASP.NET Core, то был разочарован его производительностью; оказывается, что журналирование по умолчанию замедляет работу в режиме разработки. Когда я заменил обычные средства ведения журнала и воспользовался более быстрым журналированием в памяти, страница приложения Book App, на которой отображалась информация о книге, стала в три раза быстрее! Поэтому следите за тем, чтобы слишком большое количество журналов не замедляло работу приложения.

В этой книге мы создадим приложение Book App с помощью ASP.NET Core, чтобы показать, как EF Core работает с реальными приложениями. ASP.NET Core можно использовать по-разному, но в нашем случае мы будем работать с паттерном ASP.NET Core *Модель–представление–контроллер* (MVC).

5.2 Разбираемся с архитектурой приложения Book App

В главе 2 была представлена диаграмма приложения Book App, а в главе 4 мы расширили ее, добавив еще два проекта для обработки

бизнес-логики. На рис. 5.1 показана комбинированная архитектура после главы 4 со всеми проектами в приложении. Изучая эту главу, вы узнаете, как и почему мы разбиваем код доступа к базе данных по разным проектам. Одна из причин – упростить написание, рефакторинг и тестирование веб-приложения.

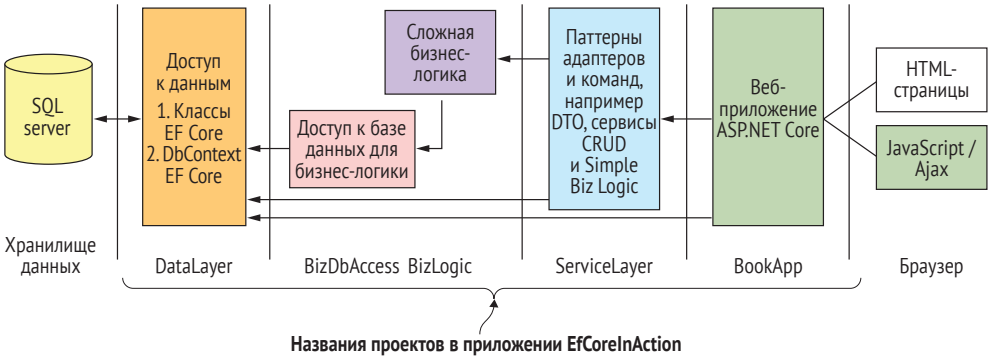


Рис. 5.1 Все проекты приложения Book App. Стрелками показаны основные маршруты, по которым данные EF Core перемещаются вверх и вниз по слоям

Многослойная архитектура, которая создает единый исполняемый файл, содержащий весь код, хорошо работает со многими поставщиками облачных услуг, которые могут запускать больше экземпляров веб-приложения, если оно находится под большой нагрузкой; ваш хост будет запускать несколько копий веб-приложения и размещать балансировщик нагрузки, чтобы распределить нагрузку по всем копиям. В Microsoft Azure этот процесс известен как *горизонтальное масштабирование*, а в Amazon Web Services он называется *автомасштабирование*.

ПРИМЕЧАНИЕ В третьей части я обновлю архитектуру приложения, чтобы использовать модульный монолит, предметно-ориентированное проектирование и чистую архитектуру. На странице http://mng.bz/5jD1 можно найти полезный документ о многослойных и чистых архитектурах.

5.3 Внедрение зависимостей

В ASP.NET Core, как и в .NET в целом, широко используется *внедрение зависимостей* (*dependency injection, DI*). Вы должны понимать его, потому что это метод, используемый в ASP.NET Core для получения экземпляра DbContext.

ОПРЕДЕЛЕНИЕ *Внедрение зависимостей* – это способ динамического связывания приложения. Обычно, чтобы создать но-

вый экземпляр `MyClass`, вы пишете: `var myClass = new MyClass()`. Такой вариант работает, но создание этого класса вшито в код, и изменить его можно, только изменив код. С помощью внедрения зависимостей можно зарегистрировать `MyClass` у поставщика внедрения зависимостей, например используя такой интерфейс, как `IMyClass`. Затем, когда вам понадобится класс, вы используете `IMyClass myClass`, а поставщик динамически создаст экземпляр и внедрит его в параметр или свойство `IMyClass myClass`.

Использование внедрения зависимостей имеет множество преимуществ, и вот основные из них:

- внедрение зависимостей позволяет приложению компоноваться динамически. Поставщик определит нужные вам классы и создаст их в правильном порядке. Например, если одному из ваших классов требуется DbContext, то внедрение зависимостей может его предоставить;
- совместное использование интерфейсов и внедрения зависимостей означает, что ваше приложение будет не так сильно связано; вы можете заменить один класс другим классом, который реализует тот же интерфейс. Этот метод особенно полезен при модульном тестировании: вы можете предоставить замену версии сервиса, используя другой, более простой класс, реализующий интерфейс (в модульных тестах это называется *заглушкой*, или *имитацией*);
- существуют и другие, более продвинутые функции, например использование внедрения зависимостей, чтобы выбрать, какой класс вернуть на основе определенных настроек. Если вы создаете приложение для электронной коммерции, то в режиме разработки вы наверняка захотите использовать фиктивный обработчик кредитных карт вместо настоящей системы кредитных карт.

Я часто использую внедрение зависимостей и не смог бы создать ни одного реального приложения без него, но признаю, что поначалу оно может сбивать с толку.

ПРИМЕЧАНИЕ В этом разделе дается краткое введение во внедрение зависимостей, чтобы вы поняли, как использовать его с EF Core. Если вам нужна дополнительная информация о внедрении зависимостей в ASP.NET Core, обратитесь к документации Microsoft на странице http://mng.bz/Kv16. Чтобы получить целостное представление, обратитесь к книге «*Внедрение зависимостей на платформе .NET*» Стивена Ван Дерсена и Марка Симанна, в которой есть целая глава, посвященная внедрению зависимостей в .NET Core.

5.3.1 Почему нужно знать, что такое внедрение зависимостей, работая с ASP.NET Core

В главе 2 было показано, как создать экземпляр DbContext с помощью следующего фрагмента кода:

```
const string connection =
     "Data Source=(localdb)\\mssqllocaldb;" +
     "Database=EfCoreInActionDb.Chapter02;" +
     "Integrated Security=True;";
var optionsBuilder =
    new DbContextOptionsBuilder
           <EfCoreContext>();

optionsBuilder.UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{…
```

Это рабочий код, но есть несколько проблем. Во-первых, вам придется повторять его для каждого подключения к базе данных. Во-вторых, в нем используется фиксированная строка доступа к базе данных: *строка подключения*. Она не будет работать, если вы захотите развернуть свой сайт на другом узле, потому что расположение базы данных на стороне сервера будет отличаться от базы данных, которую вы используете для разработки.

Можно обойти эти две проблемы несколькими способами, например переопределить метод `OnConfiguration` в DbContext (описано в разделе 5.11.1). Но лучший способ справиться с этой ситуацией – внедрение зависимостей, и это именно то, что применяет ASP.NET Core. Используя несколько иной набор команд, можно указать поставщику внедрения зависимостей, как создать DbContext – это процесс, называемый *регистрацией сервиса*, – а затем запросить экземпляр DbContext в любой подсистеме ASP.NET Core, поддерживающей внедрение зависимостей.

5.3.2 Базовый пример внедрения зависимостей в ASP.NET Core

Написание кода для конфигурирования DbContext покажется немного сложным и частично может скрывать внедрение зависимостей. В моем первом примере, показанном на рис. 5.2, используется простой класс `Demo`, который вы будете применять в контроллере ASP.NET. Этот пример будет полезен в разделе 5.7, где я покажу, как использовать внедрение зависимостей, чтобы упростить вызов кода.

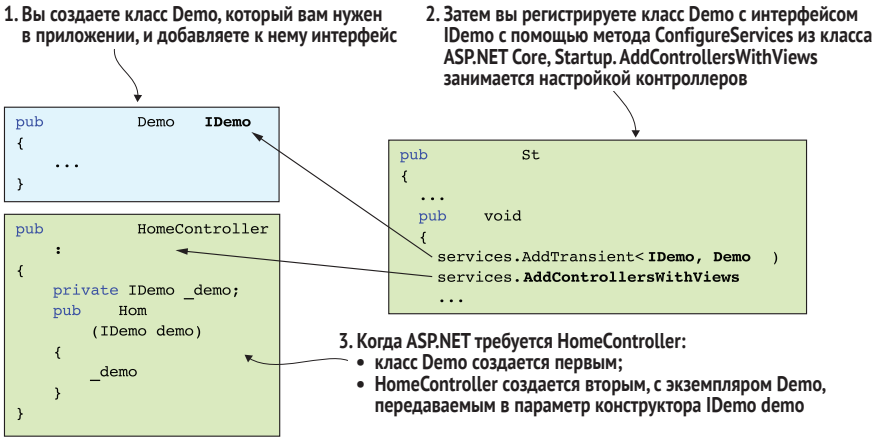


Рис. 5.2 Пример класса `Demo`, внедряемого с помощью внедрения зависимостей в конструктор контроллера. Код справа регистрирует пару `IDemo/Demo`, а команда `AddControllersWithViews` регистрирует все контроллеры ASP.NET Core. Когда ASP.NET Core требуется `HomeController` (используемый для показа HTML-страниц), внедрение зависимостей создает его. Поскольку `HomeController` требуется экземпляр `IDemo`, внедрение зависимостей создаст его и внедрит в конструктор `HomeController`

На рис. 5.2 показано, что, зарегистрировав пару `IDemo/Demo`, можно получить к ней доступ в классе `HomeController`. Зарегистрированные классы называются *сервисами*.

Правило гласит: на любой сервис, поддерживающий внедрение зависимостей, можно ссылаться или *внедрять* в любой другой сервис, поддерживающий внедрение зависимостей. На рис. 5.2 мы регистрируем наш класс `IDemo/Demo` и вызываем конфигурационный метод `AddControllersWithViews` для регистрации классов контроллера ASP.NET Core, в данном случае класса `HomeController`. Это позволяет использовать интерфейс `IDemo` в конструкторе `HomeController`, а внедрение зависимостей предоставляет экземпляр класса `Demo`. Говоря терминами внедрения зависимостей, мы используем *внедрение через конструктор* для создания экземпляра зарегистрированного нами класса. В этой главе мы будем по-разному использовать внедрение зависимостей, но определенные здесь правила и термины помогут вам разобраться в последующих примерах.

5.3.3 Жизненный цикл сервиса, созданного внедрением зависимостей

Одна из особенностей внедрения зависимостей, которая важна, когда мы говорим о EF Core, – это *жизненный цикл* экземпляра, созданного внедрением зависимостей, т. е. как долго экземпляр существует до того, как будет потерян или удален. В нашем примере с `IDemo/Demo` мы зарегистрировали экземпляр как *transient*; при каждом запросе созда-

ется новый экземпляр Demo. Если вы хотите использовать собственные классы с внедрением зависимостей, то, скорее всего, объявите жизненный цикл типа *transient*; это то, что я использую для всех своих сервисов, поскольку это означает, что каждый экземпляр запускается с настройками по умолчанию. Простые классы, сохраняющие данные, например настройки при запуске, можно объявить как *одиночки* (*singleton*) (каждый раз вы будете получать один и тот же экземпляр).

DbContext отличается тем, что его жизненный цикл относится к типу *scoped*. Это означает, что сколько бы экземпляров DbContext вы бы ни запрашивали в течение одного HTTP-запроса, вы будете получать один и тот же экземпляр. Но когда этот HTTP-запрос завершается, экземпляр исчезает (технически, поскольку DbContext реализует `IDisposable`, он удаляется), и вы получите новый экземпляр с жизненным циклом типа Scoped в следующем HTTP-запросе. На рис. 5.3 показаны три типа жизненного цикла со своей буквой для каждого нового экземпляра.

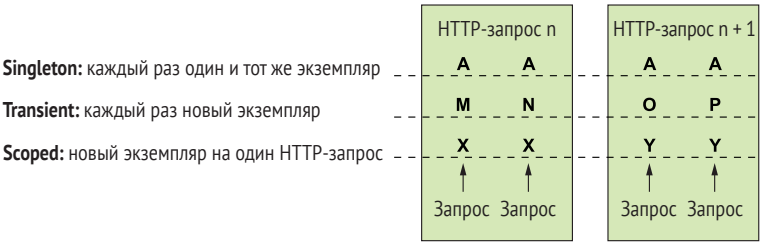


Рис. 5.3 Экземпляры, созданные внедрением зависимостей, имеют три типа жизненного цикла: singleton, transient и scoped. На этом рисунке показаны эти три типа с четырьмя внедрениями для каждого из них, по два на HTTP-запрос. Буквы обозначают экземпляр. Если буква используется несколько раз, то все эти внедрения являются одним и тем же экземпляром класса

Для DbContext необходимо использовать жизненный цикл типа scoped, если вы его внедряете в несколько классов. Иногда, например, полезно разбить сложное обновление на несколько классов. Если вы это сделаете, нужно, чтобы DbContext приложения был одинаковым для всех классов; в противном случае изменения, внесенные в один класс, не появятся в другом классе.

Разделим сложное обновление на классы `Main` и `SubPart`, где класс `Main` получает экземпляр `SubPart` через интерфейс `ISubPart` в своем конструкторе. Теперь часть `Main` вызывает метод в интерфейсе `ISubPart`, а код `SubPart` загружает класс сущности и изменяет свойство. В конце всего обновления код `Main` вызывает метод `SaveChanges`. Если DbContext двух приложений, внедренных в классы `Main` и `SubPart`, разный, то изменение, внесенное классом `SubPart`, теряется.

Эта ситуация может показаться непонятной или необычной, но даже в приложениях среднего размера она может случаться очень часто. Я нередко разбиваю сложный код на отдельные классы потому,

что весь код слишком большой, либо потому, что хочу протестировать разные части кода отдельно.

И наоборот, у каждого HTTP-запроса должен быть собственный экземпляр DbContext, поскольку DbContext в EF Core не является потокобезопасным (см. раздел 5.11.1). Именно поэтому DbContext имеет жизненный цикл типа scoped для каждого HTTP-запроса, и это одна из причин, почему внедрение зависимостей так полезно.

5.3.4 Особые соображения, касающиеся приложений Blazor Server

Если вы используете клиентское приложение на основе Blazor, взаимодействующее с серверной частью ASP.NET Core (известно как *модель размещения Blazor Server*), необходимо изменить подход к регистрации и/или получению экземпляра DbContext своего приложения. Проблема состоит в том, что, используя клиентскую часть Blazor, можно отправлять вызовы для доступа к базе данных параллельно. Это означает, что несколько потоков попытаются использовать один экземпляр DbContext, что недопустимо.

Есть несколько способов обойти эту проблему, но самый простой – создавать новый экземпляр DbContext для каждого доступа к базе данных. EF Core 5 предоставила фабричный метод DbContext, который создает новый экземпляр каждый раз, когда вы его вызываете (см. раздел 5.4.3). Фабричный метод DbContext предотвращает попытки нескольких потоков использовать один и тот же экземпляр DbContext.

Обратная сторона применения фабричного метода состоит в том, что разные классы, зарегистрированные для внедрения зависимостей, не будут использовать один и тот же экземпляр DbContext. Экземпляр DbContext с жизненным циклом типа scoped из раздела 5.3.3, например, может вызвать проблемы, потому что у классов `Main` и `SubPart` будут разные экземпляры DbContext приложения. Одним из решений данной проблемы является получение классом `Main` экземпляра DbContext приложения и передача этого экземпляра классу `SubPart` путем создания самого `SubPart` либо с помощью параметра метода.

Даже при использовании подхода с фабричным методом DbContext могут возникнуть проблемы с долгоживущими сервисами. Команда EF Core написала руководство по применению EF Core с приложением Blazor Server, предоставляя в качестве примера приложения, в котором показаны некоторые методы: http://mng.bz/yY7G.

5.4 Делаем DbContext приложения доступным, используя внедрение зависимостей

Теперь, когда вы знаете, что такое внедрение зависимостей, можно настроить DbContext своего приложения как сервис, чтобы вы могли

позже получить к нему доступ с помощью внедрения зависимостей. Это делается при запуске веб-приложения ASP.NET Core. Вы регистрируете DbContext у поставщика внедрения зависимостей, который сообщает EF Core информацию, к какой базе данных вы обращаетесь и где она расположена.

5.4.1 Предоставление информации о расположении базы данных

При разработке приложения нужно запустить его на компьютере разработчика и получить доступ к локальной базе данных для тестирования. Тип базы данных будет определяться бизнес-потребностями, но расположение базы данных на вашем компьютере зависит от вас и сервера базы данных, который вы используете.

Что касается веб-приложений, то расположение базы данных обычно не вшито в код приложения, потому что оно изменится, когда веб-приложение будет перемещено на хост, где к нему могут получить доступ реальные пользователи. Следовательно, расположение и различные параметры конфигурации базы данных обычно хранятся в виде *строки подключения*. Эта строка хранится в файле с настройками приложения, который ASP.NET читает при запуске. В ASP.NET Core есть ряд таких файлов, но пока мы сосредоточимся на трех стандартных файлах:

- *appsetting.json* – содержит настройки, общие для окружения разработки и промышленного окружения;
- *appsettings.Development.json* – содержит настройки для окружения разработки;
- *appsettings.Production.json* – содержит настройки для промышленного окружения (когда веб-приложение развертывается на хосте, чтобы пользователи могли получить к нему доступ).

ПРИМЕЧАНИЕ В этих файлах есть еще много всего, о чем мы не упомянули. Пожалуйста, обратитесь к документации по APS.NET Core, где приводится более полное описание.

Обычно строка подключения для разработки хранится в файле appsettings.Development.json. В листинге 5.1 показана строка подключения, подходящая для локального запуска базы данных SQL на ПК с Windows.

ПРИМЕЧАНИЕ Установка Visual Studio включает функцию *SQL Server Express*, что позволяет использовать SQL Server для разработки.

Листинг 5.1 Файл appsettings.Development.json со строкой подключения к базе данных

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=EfCoreInActionDb
;Trusted_Connection=True"
  },
  … Другие части кода были удалены, поскольку они не относятся к доступу к базе
данных;
}
```

Необходимо отредактировать файл appsettings.Development.json, чтобы добавить строку подключения для вашей локальной базы данных. В этом файле может быть или не быть раздела `ConnectionStrings`, в зависимости от того, настроена ли проверка подлинности для отдельных учетных записей пользователей. (Для параметра «Отдельные учетные записи пользователей» требуется собственная база данных, поэтому Visual Studio добавляет строку подключения для базы данных авторизации в файл appsetting.json.) Вы можете назвать свою строку подключения как угодно; в этом примере в нашем приложении используется имя `DefaultConnection`.

5.4.2 Регистрация DbContext приложения у поставщика внедрения зависимостей

Следующий шаг – регистрация DbContext у поставщика внедрения зависимостей при запуске. Любая конфигурация, которая должна быть выполнена при запуске ASP.NET Core, выполняется в классе с метким названием `Startup`. Этот класс выполняется при запуске приложения ASP.NET Core и содержит несколько методов для установки и настройки веб-приложения.

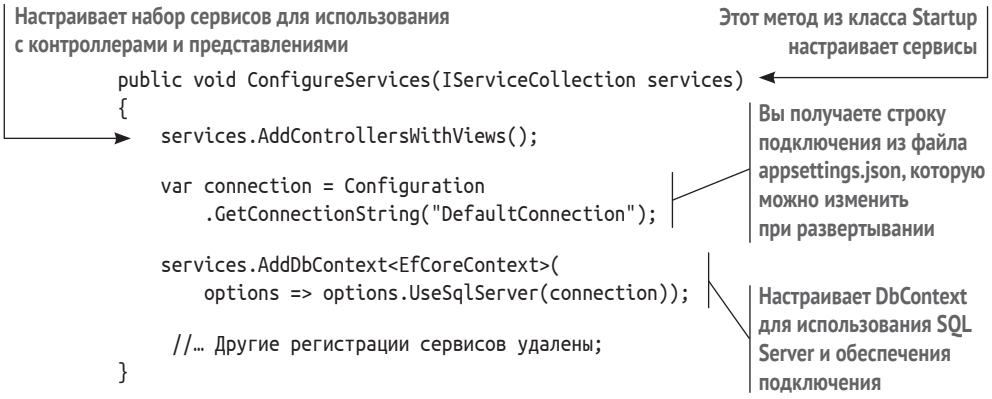
У DbContext для ASP.NET Core есть конструктор, который принимает параметр `DbContextOptions<T>`, определяющий параметры базы данных. Таким образом, строка подключения к базе данных может измениться при развертывании веб-приложения (см. раздел 5.8).

Напоминаю, вот как выглядит конструктор DbContext в приложении Book App, показанный жирным шрифтом в этом фрагменте кода:

```
public class EfCoreContext : DbContext
{
    //… Свойства были удалены для ясности;

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

    //… Остальной код был удален для ясности;
}
```

В следующем листинге показано, что DbContext регистрируется как сервис в приложении ASP.NET Core. Эта регистрация выполняется в методе `ConfigureServices` в классе `Startup` вашего приложения ASP.NET Core вместе со всеми сервисами, которые вам необходимо зарегистрировать.

Листинг 5.2 Регистрация DbContext в классе ASP.NET Core `Startup`




```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    var connection = Configuration
        .GetConnectionString("DefaultConnection");

    services.AddDbContext<EfCoreContext>(
        options => options.UseSqlServer(connection));

     //… Другие регистрации сервисов удалены;
}
```

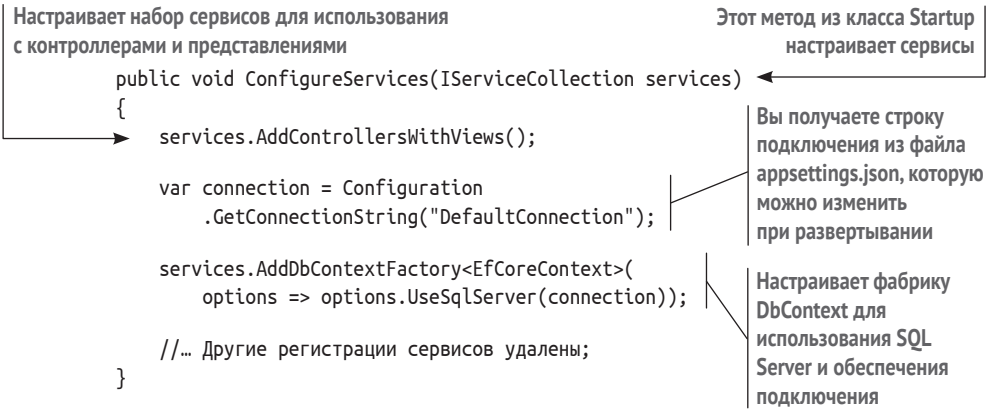
Первый шаг – получение строки подключения из класса `Configuration`. В ASP.NET Core он настраивается во время работы конструктора класса `Startup`, который считывает файлы appsetting. Получая его таким образом, можно изменить строку подключения к базе данных при развертывании приложения на хосте. В разделе 5.8.1, посвященном развертыванию приложения ASP.NET Core, использующего базу данных, описано, как работает этот процесс.

Второй шаг – обеспечение доступности DbContext через внедрение зависимостей – выполняется методом `AddDbContext`, который регистрирует экземпляры DbContext, `EfCoreContext` и `DbContextOptions<EfCoreContext>` как сервисы. Когда вы используете тип `EfCoreContext` в местах, где внедрение зависимостей выполняет перехват, поставщик создает экземпляр DbContext, используя `DbContextOptions<EfCoreContext> options`. Или, если вы запрашиваете несколько экземпляров в одном том же HTTP-запросе, поставщик вернет те же самые экземпляры. Вы увидите этот процесс в действии, когда начнете использовать DbContext для выполнения запросов к базе данных и обновлений в разделе 5.6.

5.4.3 Регистрация фабрики DbContext у поставщика внедрения зависимостей

Как сказано в разделе 5.3.4, приложениям Blazor Server и некоторым другим типам приложений необходимо тщательное управление экземплярами DbContext. В EF Core 5 был добавлен интерфейс `IDbContextFactory<TContext>` наряду с методом для регистрации фабрики DbContext, как показано в следующем листинге.

Листинг 5.3 Регистрация фабрики DbContext в классе ASP.NET Core Startup



Обычно метод `AddDbContextFactory` используется только с Blazor в клиентской части или в приложениях, где нельзя контролировать параллельный доступ к одному и тому же DbContext, что нарушает правило безопасности потоков (см. раздел 5.11.1). Многие другие приложения, например приложения ASP.NET Core, управляют параллельным доступом за вас, поэтому вы можете получить экземпляр DbContext, используя внедрение зависимостей.

5.5 Вызов кода доступа к базе данных из ASP.NET Core

Настроив DbContext и зарегистрировав его как сервис, можно получить доступ к базе данных. В этих примерах мы выполним запрос для отображения книг и команды для обновления базы данных. Мы сосредоточимся на том, как выполнять эти методы из ASP.NET Core; полагаю, вы уже поняли, как выполнять запросы и обновлять базу данных, из предыдущих глав.

> **ПРИМЕЧАНИЕ** Пример кода в основном касается использования паттерна ASP.NET Core, MVC, но все примеры использования внедрения зависимостей также применимы ко всем формам ASP.NET Core: страницы Razor, MVC и Web-API. В нескольких разделах рассматриваются приложения Blazor Server, потому что обработка получения экземпляра DbContext приложения выглядит иначе.

5.5.1 Краткое изложение того, как работает паттерн ASP.NET Core MVC, и термины, которые он использует

Во-первых, вот краткое описание того, как использовать ASP.NET Core для реализации нашего приложения Book App. Для создания различных HTML-страниц мы будем использовать *контроллер* (*controller*) ASP.NET Core, представляющий собой класс, который обрабатывает представление HTML-страниц с помощью представлений Razor. Для этого мы создадим класс `HomeController`, который наследует от класса ASP.NET Core `Controller`. У этого контроллера есть несколько представлений Razor, связанных с его методами, которые в ASP.NET Core известны как *методы действий* (*action method*).

У `HomeController` приложения Book App есть метод действия `Index`, который показывает список книг, и метод `About`, предоставляющий сводную страницу для сайта. Есть и другие контроллеры для обработки оформления покупок, существующих заказов, администрирования и т. д. Хотя и можно поместить весь код доступа к базе данных в каждый метод действия каждого контроллера, я редко это делаю, потому что использую принцип разработки программного обеспечения под названием *разделение ответственностей* (*Separation of Concerns, SoC*). О нем рассказывается в следующем подразделе.

5.5.2 Где находится код EF Core в приложении Book App?

Как вы знаете из раздела 5.2, наше приложение Book App создано с применением многослойной архитектуры. Речь идет об архитектуре, которую можно использовать в реальном приложении.

В этом разделе вы увидите, где разместить различные части кода доступа к базе данных EF Core и почему.

> **ОПРЕДЕЛЕНИЕ** *Разделение ответственностей* – это идея, согласно которой программная система должна быть разложена на части, которые как можно реже пересекаются по функциональности. Это описывается двумя другими принципами: связанность и связность. Для слабой *связанности* нужно, чтобы каждый проект в приложении был как можно более самодостаточным, а для *связности* каждый проект в приложении должен содержать код, предоставляющий близкие по смыслу функции. Посетите страницу http://mng.bz/wHJS для получения дополнительной информации.

На рис. 5.4 показано расположение кода доступа к базе данных в приложении на ранее использованной схеме архитектуры (рис. 5.1). Круги показывают, какой тип кода базы данных вы найдете в каждом слое. Обратите внимание, что в проекте ASP.NET Core и проекте чистой бизнес-логики (BizLogic) нет кода запросов / обновления EF Core.

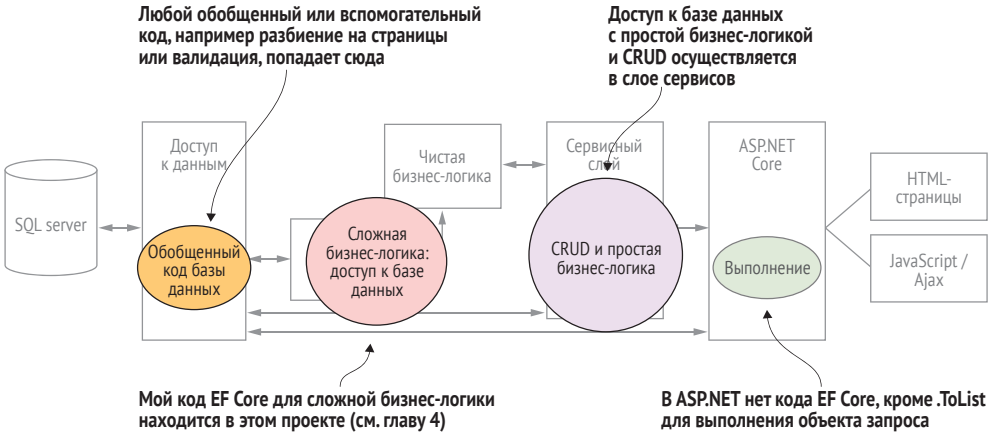


Рис. 5.4 Расположение кода доступа к базе данных (код EF Core) в приложении Book App. Такое разделение кода EF Core упрощает поиск, восприятие, рефакторинг и тестирование

Применение принципов разделения ответственностей имеет ряд преимуществ для всего приложения. В главе 4 вы узнали о причине разделения сложной бизнес-логики. Но в этой главе вы увидите преимущества для проекта ASP.NET Core:

- клиентская часть ASP.NET Core предназначена для отображения данных, и сделать это хорошо – серьезная задача, требующая большой концентрации. Следовательно, вы будете использовать сервисный слой для обработки команд EF Core и преобразования данных из базы данных в форму, которую может легко использовать клиентская часть ASP.NET Core – часто через объекты DTO, также известные как модели представления (ViewModel) в ASP.NET Core. После этого можно сосредоточиться на обеспечении лучшего взаимодействия с пользователем, а не думать о том, правильно ли вы выполняете запрос к базе данных;
- у контроллеров ASP.NET часто имеется несколько страниц или действий (например, для перечисления элементов, для добавления нового элемента, для редактирования элемента и т. д.), для каждой из которых требуется свой код базы данных. Переместив код базы данных в сервисный слой, можно создавать отдельные классы для доступа к базе данных, а не распространять код по всему контроллеру;
- гораздо проще провести модульное тестирование кода базы данных, если он находится в сервисном слое, чем когда он находится в контроллере ASP.NET Core. Вы можете тестировать контроллеры ASP.NET Core, но тестирование может стать сложным, если ваш код обращается к таким свойствам, как `HttpRequest` (что он и делает), потому что некоторые из этих функций сложно воспроизвести, чтобы ваш модульный тест заработал.

Вы можете запускать тесты для своего полноценного приложения ASP.NET Core, используя пакет NuGet `Microsoft.AspNetCore.Mvc.Test-`

ing. Это тестирование известно как интеграционное, когда тестируется все приложение, в отличие от модульного тестирования, которое фокусируется на тестировании небольших частей приложения. Более подробную информацию об интеграционном тестировании можно найти на странице http://mng.bz/MXa7.

5.6 Реализация страницы запроса списка книг

Теперь, когда все готово, мы реализуем работу со списком книг из приложения Book App. Чтобы напомнить вам, как выглядит сайт, на рис. 5.5 показан скриншот приложения со списком книг и функциями администратора.

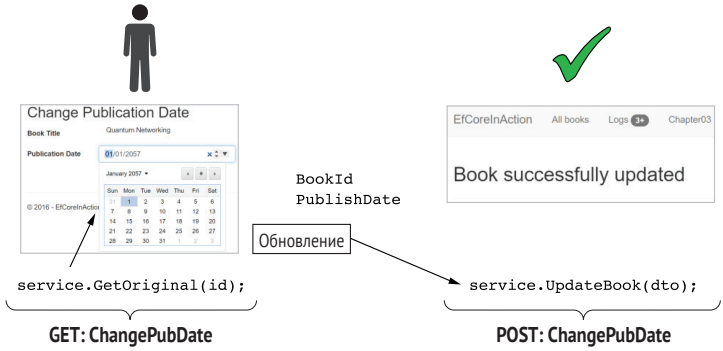


Рис. 5.5 Домашняя страница приложения Book App со списком книг и функциями администратора, включая изменение даты публикации книги

В главе 2 мы написали класс `ListBooksService`, который справился со сложностями преобразования, сортировки, фильтрации и разбиения книг на страницы для отображения. Нам понадобится использовать этот класс в действии ASP.NET Core, `Index`, в контроллере `HomeController`. Главным препятствием для создания экземпляра класса ListBookService является то, что вам потребуется получить экземпляр DbContext.

5.6.1 Внедрение экземпляра DbContext приложения через внедрение зависимостей

Стандартный способ предоставления экземпляра DbContext приложению ASP.NET Core (и другим типам размещаемых приложений) – это внедрение зависимостей через конструктор класса (см. раздел 5.3.2). Для приложения ASP.NET Core вам, возможно, потребуется добавить конструктор в контроллер, которому в качестве параметра передается экземпляр DbContext (внедрение зависимости через конструктор).

В листинге 5.4 показано начало `HomeController`, где мы добавили конструктор и сохранили `EfCoreContext` в поле класса, которое можно использовать для создания экземпляра класса `BookListService`, необходимого для того, чтобы показать список книг. В этом коде используется подход с внедрением зависимостей из раздела 5.3.2, показанным на рис. 5.2, но в данном случае класс `Demo` заменяется классом `EfCoreContext`.

Листинг 5.4 Действие `Index` в `HomeController` отображает список книг

```
public class HomeController : Controller
{
    private readonly EfCoreContext _context;

    public HomeController(EfCoreContext context)
    {
        _context = context;
    }

    public IActionResult Index
        (SortFilterPageOptions options)
    {
        var listService =
            new ListBooksService(_context);

        var bookList = listService
            .SortFilterPage(options)
            .ToList();

        return View(new BookListCombinedDto
            (options, bookList));
    }
```

DbContext предоставляется ASP.NET Core через внедрение зависимостей

Параметр options заполняется параметрами сортировки, фильтрации и разбиения на страницы через URL-адрес

Действие ASP.NET, вызываемое, когда домашняя страница вызывается пользователем

ListBooksService создается с использованием DbContext из поля _context

Метод SortFilterPage вызывается с параметрами сортировки, фильтрации и разбиения на страницы

Отправляет параметры (для заполнения элементов управления вверху страницы) и список BookListDtos для отображения в виде HTML-таблицы

Метод ToList() выполняет команды LINQ, в результате чего EF Core превращает LINQ в соответствующий SQL-код для доступа к базе данных и возвращает результат в виде списка

После того как вы использовали локальную копию DbContext для создания `ListBooksService`, можно вызвать его метод `SortFilterPage`. Этот метод принимает параметры, возвращаемые различными элементами управления на странице списка, и возвращает тип `IQueryable<BookListDto>`. Затем вы вызываете метод `ToList` в конце цепочки вызовов, что заставляет EF Core преобразовать `IQueryable` в запрос для базы данных и вернуть список с информацией о книге, которую запросил пользователь. Этот результат передается в представление ASP.NET Core для отображения.

Можно было бы заставить метод `SortFilterPage` вернуть результат `List<BookListDto>`, но этот подход ограничил бы вас использованием синхронного доступа к базе данных. Как вы увидите в разделе 5.10, возвращая результат `IQueryable<BookListDto>`, можно использовать

обычную (синхронную) или асинхронную версию финальной команды, выполняющей запрос.

5.6.2 Использование фабрики DbContext для создания экземпляра DbContext

В некоторых приложениях, таких как приложения Blazor Server (см. раздел 5.3.4), выбор стандартной области видимости DbContext не работает. В этом случае можно внедрить `IDbContextFactory<TContext>` с помощью внедрения зависимостей. Такое разделение полезно для приложений Blazor, в которых EF Core рекомендует использовать `IDbContextFactory`, и может быть полезно в других сценариях.

Вот пример, взятый из BlazorServerEFCoreSample, предоставленный командой EF Core. В этом примере `DbContextFactory` внедряется на страницу Razor в Blazor, как показано в следующем листинге. Комментарии указаны только при использовании `DbContextFactory` и создании DbContext.

Листинг 5.5 Пример внедрения фабрики DbContext на страницу Razor

```
@page "/add"

@inject IDbContextFactory<ContactContext> DbFactory
@inject NavigationManager Nav
@inject IPageHelper PageHelper
  @if (Contact != null)
  {
      <ContactForm Busy="@Busy"
                   Contact="@Contact"
                   IsAdd="true"
                   CancelRequest="Cancel"
                   ValidationResult=
  "@(async (success) => await ValidationResultAsync(success))" />
  }
  @if (Success)
  {
      <br />
      <div class="alert alert-success">The contact was successfully
      added.</div>
  }
  @if (Error)
  {
      <br />
      <div class="alert alert-danger">Failed to update the contact
       (@ErrorMessage).</div>
  }

  @code {
      //… Некоторые поля пропущены;
      private async Task ValidationResultAsync(bool success)
```

Фабрика DbContext внедряется на страницу Razor

```
    {
        if (Busy)
            return;
        if (!success)
        {
            Success = false;
            Error = false;
            return;
        }

        Busy = true;

        using var context = DbFactory.CreateDbContext();
        context.Contacts.Add(Contact);

        try
        {
            await context.SaveChangesAsync();
            Success = true;
            Error = false;
            // подготовка к следующему
            Contact = new Contact();
            Busy = false;
        }
        catch (Exception ex)
        {
            Success = false;
            Error = true;
            ErrorMessage = ex.Message;
            Busy = false;
        }
    }

    private void Cancel()
    {
        Nav.NavigateTo($"/{PageHelper.Page}");
    }
}
```

Еще один метод обработки приложений Blazor Server. Он не будет обрабатывать дополнительные запросы, пока не завершится первый запрос

Создает новый экземпляр DbContext. Обратите внимание на использование using для освобождения

Новая контактная информация добавляется в DbContext

Сохраняет Contact в базе данных

Обратите внимание, что экземпляры DbContext, которые созданы фабрикой DbContext Factory, не управляются поставщиком сервисов приложения и, следовательно, должны быть освобождены приложением. На странице Razor, показанной в листинге 5.5, `using var context =…` освободит экземпляр DbContext при выходе из области локальной переменной контекста.

ПРИМЕЧАНИЕ Страницу Razor, показанную в листинге 5.5, можно найти по адресу https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/blazor/samples/5.0/BlazorServerEFCoreSample/BlazorServerDbContextExample/Pages/AddContact.razor.

5.7 Реализация методов базы данных как сервиса внедрения зависимостей

Хотя подход с внедрением через конструктор, который мы использовали в предыдущем разделе, работает, существует еще один способ использования внедрения зависимостей, который обеспечивает лучшую изоляцию кода доступа к базе данных: *внедрение через параметр*. В ASP.NET Core можно организовать внедрение сервиса в *метод действия* с помощью параметра, помеченного атрибутом `[FromServices]`. Вы можете предоставить определенный сервис, который нужен каждому методу действия в своем контроллере; этот подход более эффективен и проще для модульного тестирования. Чтобы увидеть, как это работает, мы используем класс `ChangePubDateService`, который находится в сервисном слое, чтобы обновить дату публикации книги. Этот класс позволяет пользователю с правами администратора изменять дату публикации, как показано на рис. 5.6.

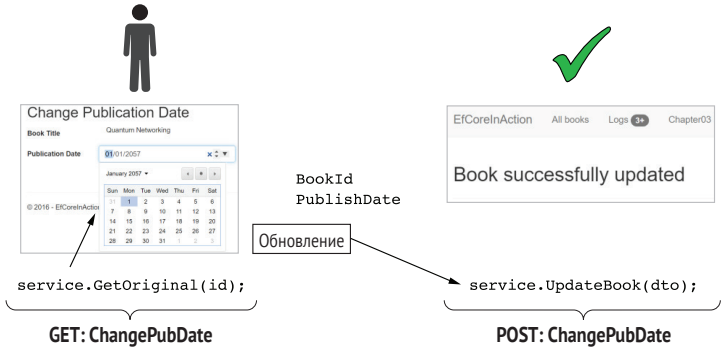


Рис. 5.6 Два этапа изменения даты публикации книги. На первом этапе с запросом `GET` вызывается метод `GetOriginal`, чтобы показать пользователю книгу и текущую дату ее публикации. Затем на этапе с запросом `POST` вызывается метод `UpdateBook` с установленной пользователем датой

Как видите, этот процесс состоит из двух этапов:

- вы показываете пользователю с правами администратора текущую дату публикации и позволяете ему изменить ее;
- к базе данных применяется обновление, и вы сообщаете пользователю, что оно было успешным.

Чтобы использовать внедрение через параметры для класса `ChangePubDateService`, необходимо сделать две вещи:

- зарегистрировать свой класс `ChangePubDateService` с помощью внедрения зависимостей, чтобы он стал сервисом, который можно внедрить, используя внедрение зависимостей;

- использовать внедрение через параметр, чтобы внедрить экземпляр класса `ChangePubDate` в два метода действия ASP.NET, которые в нем нуждаются (`GET` и POST).

Этот подход хорошо работает для создания приложений ASP.NET Core, и я использовал его во всех своих MVC-проектах ASP.NET на протяжении многих лет. Он не только обеспечивает хорошую изоляцию и упрощает тестирование, но и значительно упрощает написание методов действий контроллера ASP.NET Core. В разделе 5.7.2 вы увидите, что код внутри метода действия `ChangePubDate` простой и короткий.

5.7.1 Регистрация класса в качестве сервиса во внедрении зависимостей

В ASP.NET можно зарегистрировать класс с помощью внедрения зависимостей множеством способов. Стандартный способ – добавить в класс интерфейс `IChangePubDateService`. Технически интерфейс не нужен, но его использование – хорошая практика и может быть полезно при модульном тестировании. Кроме того, мы используем интерфейс из раздела 5.7.3, чтобы упростить регистрацию классов.

В следующем листинге показан интерфейс `IChangePubDateService`. Не забывайте, что контроллер ASP.NET Core будет иметь дело с интерфейсом `IChangePubDateService`, поэтому нужно убедиться, что объявлены все открытые методы и свойства.

Листинг 5.6 Интерфейс `IChangePubDateService`, необходимый для регистрации класса

```
public interface IChangePubDateService
{
    ChangePubDateDto GetOriginal(int id);
    Book UpdateBook(ChangePubDateDto dto);
}
```

Затем мы регистрируем этот интерфейс или класс в сервисе внедрения зависимостей. Сделать это по умолчанию в ASP.NET Core можно, добавив строку к методу `ConfigureServices` в классе `Startup`. В этом листинге показан обновленный метод, в котором новый код выделен жирным шрифтом. Мы добавляем `ChangePubDateService` как transient, потому что нам нужно, чтобы при каждом запросе создавалась новая версия.

Листинг 5.7 Метод ASP.NET Core `ConfigureService` в классе `Startup`

```
public void ConfigureServices (IServiceCollection services)
{
  // Добавляем сервисы фреймворка;
  services.AddControllersWithViews();
  var connection = Configuration
      .GetConnectionString("DefaultConnection");
```

```
    services.AddDbContext<EfCoreContext>(
        options => options.UseSqlServer(connection))

    services.AddTransient
        <IChangePubDateService, ChangePubDateService>();
}
```

Регистрирует класс ChangePubDateService как сервис с интерфейсом IChangePubDateService в качестве способа доступа к нему

5.7.2 Внедрение ChangePubDateService в метод действия ASP.NET

Настроив класс `ChangePubDateService` как сервис, который можно внедрить, используя внедрение зависимостей, теперь нужно создать экземпляр в `AdminController`. Оба метода действия ASP.NET Core называются `ChangePubDate`; один метод – `GET` для заполнения страницы редактирования, а второй – `POST` для обновления.

На рис. 5.7 показано, как, используя внедрение зависимостей, создать сервис `ChangePubDateService`, в который через конструктор внедряется экземпляр `EfCoreDbContext`. Затем сервис `ChangePubDate` внедряется в действие `GET AdminController` посредством внедрения через параметры.

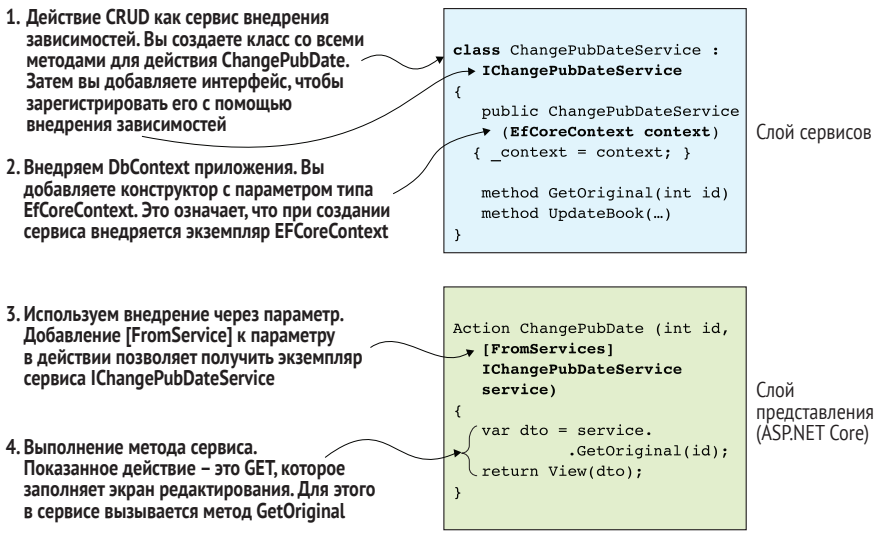


Рис. 5.7 Использование внедрения зависимостей для предоставления сервиса часто требует, чтобы поставщик внедрения зависимостей сначала создал другие классы. В этом, довольно простом случае есть как минимум четыре уровня внедрения зависимостей. Вызывается метод `ChangePubDate` `AdminController` (нижний прямоугольник); затем атрибут `[FromServices]` одного из параметров метода велит поставщику создать экземпляр класса `ChangePubDateService`. Классу `ChangePubDateService` (верхний прямоугольник) требуется экземпляр класса `EfCoreDbContext`, поэтому поставщик также должен создать этот экземпляр, что, в свою очередь, требует создания `DbContextOptions<EfCoreContext>`, чтобы можно было создать класс `EfCoreDbContext`

Как вы увидите, поставщик внедрения зависимостей вызывается много раз для создания всех классов, необходимых для обработки HTTP-запроса.

Можно было бы предоставить экземпляр класса `ChangePubDateService` с помощью внедрения через конструктор, как мы это делали с DbContext, но у этого подхода есть и обратная сторона. `AdminController` содержит несколько команд обновления базы данных, например добавление отзыва на книгу, рекламного предложения и т. д. Использование внедрения через конструктор означало бы, что мы без необходимости создавали бы экземпляр класса `ChangePubDateService` при вызове любой из этих команд. Используя внедрение через параметр в каждое действие, вы тратите время и память на создание только нужного вам сервиса. В следующем листинге показано действие `GET`, `ChangePubDate`, которое вызывается, когда кто-то щелкает по ссылке «Администратор»> «Изменить дату публикации», желая изменить дату публикации.

Листинг 5.8 Метод действия `ChangePubDate` в `AdminController`

Действие вызывается, если пользователь щелкает по ссылке «Администратор» > «Изменить дату публикации»

Получает первичный ключ книги, которую пользователь хочет изменить

```
public IActionResult ChangePubDate
    (int id,
    [FromServices]IChangePubDateService service)
{
    var dto = service.GetOriginal(id);
    return View(dto);
}
```

Внедряет экземпляр ChangePubDateService

Использует сервис для получения DTO, чтобы показать его пользователю

Показывает страницу, позволяющую пользователю редактировать дату публикации

Строка 3 (выделенная жирным шрифтом) в этом листинге является важной. Мы использовали внедрение через параметр, чтобы внедрить экземпляр класса `ChangePubDateService`. Эта же строка находится в `POST`-версии действия `ChangePubDate`.

Обратите внимание, что классу `ChangePubDateService` требуется класс `EfCoreContext`, который является DbContext приложения, в конструкторе. Это нормально, потому что внедрение зависимостей рекурсивно; оно будет продолжать заполнять параметры значениями или осуществлять другие внедрения, пока все необходимые классы не будут зарегистрированы.

5.7.3 Улучшаем регистрацию классов доступа к базе данных как сервисов

Прежде чем закончить с темой внедрения зависимостей, хочу описать более подходящий способ регистрации классов как сервисов.

В предыдущем примере, в котором вы превратили класс `ChangePubDateService` в сервис, требовалось добавить код для регистрации этого класса как сервиса в `ConfigureServices`. Этот процесс работает, но требует много времени и подвержен ошибкам, поскольку нужно добавить строку кода для регистрации каждого класса, который вы хотите использовать в качестве сервиса.

В первом издании этой книги я предлагал использовать библиотеку внедрения зависимостей Autofac (https://autofaccn.readthedocs.io/en/latest), потому что в ней есть команда, которая регистрирует все классы с интерфейсами в сборке (также известной как *проект*). Позже я наткнулся на твит Дэвида Фаулера, который ссылается на набор тестов для контейнеров внедрения зависимостей, – http://mng.bz/go2l. На этой странице я узнал, что контейнер внедрения зависимостей ASP.NET Core намного быстрее, чем AutoFac! На данном этапе я создал библиотеку под названием NetCore.AutoRegisterDi (http://mng.bz/5jDz), у которой только одна задача: зарегистрировать все классы с интерфейсами в сборке с помощью поставщика внедрения зависимостей .NET Core.

> **ПРИМЕЧАНИЕ** После того как я создал свою библиотеку `NetCore.AutoRegisterDi`, Эндрю Лок указал мне на существующую библиотеку под названием Scrutor; см. его статью на странице http://mng.bz/6gly. Scrutor имеет больше возможностей для выбора классов для регистрации, чем `NetCore.AutoRegisterDi`, поэтому вам стоит взглянуть на нее.

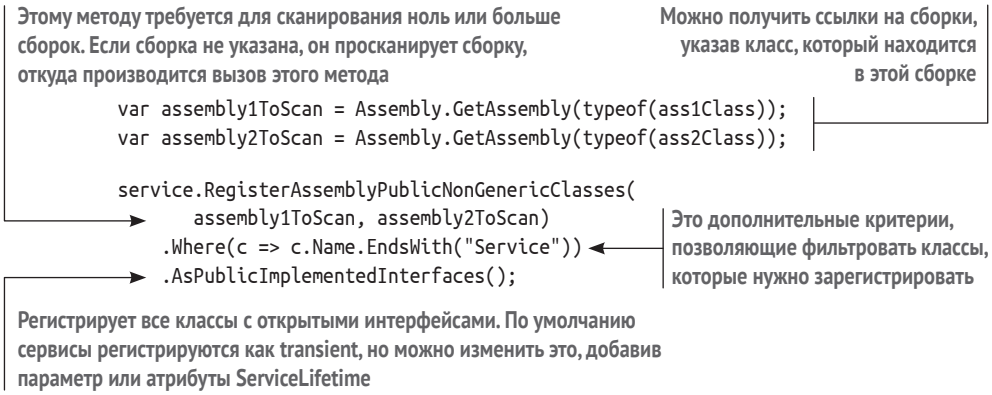
КАК У МЕНЯ ОРГАНИЗОВАНА РЕГИСТРАЦИЯ СЕРВИСОВ В КОНТЕЙНЕРЕ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ NET CORE

Библиотека `NetCore.AutoRegisterDi` проста: она сканирует одну или несколько сборок; ищет стандартные открытые необобщенные классы с открытыми интерфейсами и регистрирует их в поставщике внедрения зависимостей NET Core. У нее простая фильтрация и есть возможности настройки жизненного цикла, но не более того (всего примерно 80 строк кода). Но этот простой кусок кода дает два преимущества по сравнению с ручной регистрацией классов или интерфейсов у поставщика внедрения зависимостей:

- он экономит ваше время, потому что вам не нужно регистрировать каждый интерфейс или класс вручную;
- что еще более важно, он автоматически регистрирует ваши интерфейсы или классы, чтобы вы ничего не забыли.

Вторая причина, почему я считаю эту библиотеку такой полезной: я никогда не забуду зарегистрировать сервис. В следующем листинге показан типичный вызов библиотеки `NetCore.AutoRegisterDi`.

Листинг 5.9 Использование библиотеки `NetCore.AutoRegisterDi` для регистрации классов как сервисов внедрения зависимостей

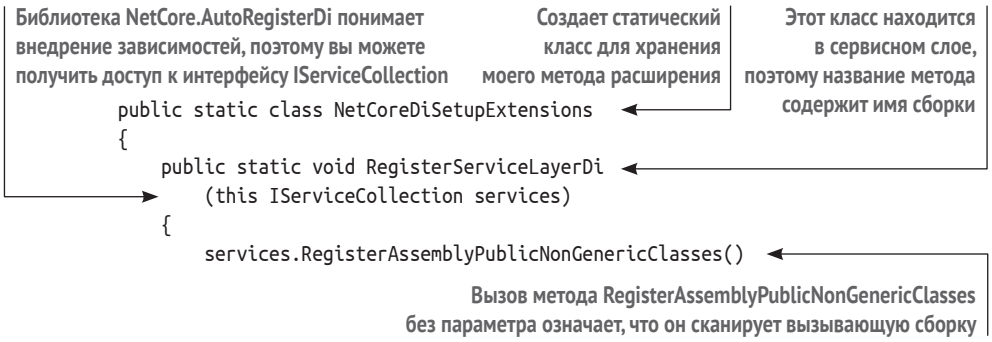


Я мог бы использовать вызов, подобный тому, что показан в листинге 5.9, в методе `Configure` из класса `Startup`, который регистрирует все сборки, но не делаю этого. Я предпочитаю добавлять метод расширения в каждый проект, где есть классы, которые необходимо зарегистрировать как сервисы внедрения зависимостей. Таким образом, я выделяю настройку каждого проекта в отдельный класс в каждом проекте, который в нем нуждается.

Каждый метод расширения использует библиотеку `NetCore.AutoRegisterDi` для регистрации стандартных классов или сервисов в проекте. В методе расширения также есть место для дополнительного кода, например для регистрации классов или сервисов, которые нельзя зарегистрировать автоматически, например обобщенные классы или сервисы.

В следующем листинге показан пример метода расширения в сервисном слое. Этот код требует добавления пакета NuGet `NetCore.AutoRegisterDi` в данный проект.

Листинг 5.10 Метод расширения в сервисном слое, обрабатывающий всю регистрацию сервисов



```
            .AsPublicImplementedInterfaces();
    }
}
```

Этот метод зарегистрирует все открытые классы с интерфейсами с жизненным циклом типа transient

Для регистрации, написанной вручную, которую NetCore.AutoRegisterDi не может осуществить, например для обобщенных классов

Приложение Book App из первой части книги содержит классы и сервисы, которые необходимо зарегистрировать в проектах ServiceLayer, BizDbAccess и BizLogic. Для этого мы копируем код из листинга 5.10 в другие проекты и меняем имя метода, чтобы каждый из них можно было идентифицировать. При вызове каждого метода `RegisterAssemblyPublicNonGenericClasses` по умолчанию сканирует сборку, из которой он вызван, и автоматически регистрирует стандартные сервисы.

Теперь, когда у вас есть отдельные версии листинга 5.8 для каждого из трех проектов, необходимо вызвать каждый из этих методов регистрации, чтобы настроить сервис внедрения зависимостей. Для этого нужно добавить следующий код в метод `ConfigureServices` в классе ASP.NET Core `Startup`.

Листинг 5.11 Вызов всех наших методов регистрации в проектах, которым они нужны

```
public void ConfigureServices(IServiceCollection services)
{
    //… Другие регистрации не указаны;

    services.RegisterBizDbAccessDi();
    services.RegisterBizLogicDi();
    services.RegisterServiceLayerDi();
}
```

Этот метод из класса Startup настраивает сервисы для ASP.NET Core

Здесь мы добавляем свои методы расширения для регистрации

В результате все классы, которые мы написали с открытыми интерфейсами в проектах ServiceLayer, BizDbAccess и BizLogic, будут автоматически зарегистрированы как сервисы внедрения зависимостей.

5.8 Развертывание приложения ASP.NET Core с базой данных

После завершения разработки приложения ASP.NET Core с базой данных в какой-то момент вам понадобится скопировать его на веб-сервер, чтобы другие могли его использовать. Этот процесс называется *развертыванием* приложения на *хосте*. В данном разделе показано, как это сделать.

ПРИМЕЧАНИЕ Для получения дополнительной информации о развертывании обратитесь к книге ASP.NET Core Эндрю Лока «*ASP.NET Core в действии*», где есть глава, посвященная развертыванию; или воспользуйтесь интерактивной документацией Microsoft на странице http://mng.bz/op7M.

5.8.1 Местонахождение базы данных на веб-сервере

Когда вы запускаете приложение ASP.NET Core локально, оно обращается к серверу базы данных на вашем компьютере. В этом примере используется Visual Studio, который поставляется с локальным SQL-сервером для разработки, доступным по ссылке `(localdb)\mssqllocaldb`. Как объяснялось в разделе 5.4.1, строка подключения для этой базы данных хранится в файле appsettings.Development.json.

Когда вы развертываете свое приложение на веб-сервере, Visual Studio по умолчанию перестраивает его с переменной `ASPNETCORE_ENVIRONMENT`, для которой установлено значение `Production`. Такая настройка заставляет приложение сначала попытаться загрузить файл appsetting.json, а затем файл appsettings.Production.json. Файл appsettings.Production.json – это то место, куда вы (или система публикации) помещаете строку подключения для базы данных вашего хоста.

СОВЕТ При запуске файл appsettings.Production.json считывается последним и отменяет любые настройки с таким же именем в файле appsetting.json. Таким образом, если вы хотите, можно поместить настройку строки подключения для разработки в файл appsetting.json, но все же лучше всего поместить ее в файл appsettings.Development.json.

Можно задать строку подключения для размещенной на хосте базы данных вручную с помощью функции публикации Visual Studio; щелкните правой кнопкой мыши проект ASP.NET Core в представлении «Обозреватель решений» и выберите **Publish** (Опубликовать). Когда вы публикуете приложение, Visual Studio создает/обновляет файл appsettings.Production.json с указанной вами строкой подключения и развертывает этот файл вместе с приложением. При запуске конструктор класса ASP.NET Core `Startup` считывает оба файла, и используется строка подключения appsettings.Production.json.

Большинство хостинговых систем Windows предоставляют профиль публикации Visual Studio, который можно импортировать в функцию публикации. Этот профиль значительно упрощает настройку развертывания, поскольку он не только подробно описывает, куда должно быть записано приложение ASP.NET Core, но и предоставляет строку подключения для размещенной базы данных.

В облачных системах, таких как Azure Web App Service, есть функция, которая может переопределять свойства в файле appsettings.json при развертывании. Это означает, что вы можете настроить соедине-

ние с базой данных, задав имя пользователя и пароль базы данных в Azure; ваше имя пользователя и пароль не находятся в среде разработки и, следовательно, лучше защищены.

5.8.2 Создание и миграция базы данных

Когда ваше приложение и его база данных работают на веб-сервере, управление базой данных меняется. На компьютере, используемом для разработки, можно делать с базой данных практически все, что угодно, но после развертывания на веб-сервере правила могут измениться. То, что вы можете делать с базой данных, зависит от хоста или бизнес-правил вашей компании.

Например, версия приложения Book App из первого издания этой книги была размещена на рентабельной (дешевой!) платформе общего хостинга (WebWiz), что не позволяло приложению создавать или удалять базу данных. Кроме этого, я использовал облачную систему Microsoft Azure, в которой могу удалять и создавать базу данных, но создание базы данных занимает много времени.

Самый простой подход, который работает во всех системах, с которыми я сталкивался, – заставить систему хостинга создать пустую базу данных, а затем применить команды для изменения ее структуры. Самый простой способ – сделать это с помощью миграции, о которой я сейчас расскажу, но есть и другие способы.

> **ПРЕДУПРЕЖДЕНИЕ** Прежде чем начать, должен предупредить вас, что к изменению структуры базы данных сайта необходимо подходить осторожно, особенно если это касается сайтов, работающих круглосуточно и без выходных, которые должны продолжать работать во время изменения базы данных. Многое может пойти не так, и это может привести к потере данных или вывести сайт из строя.

В этой главе описывается система миграций. Это хорошая система, но у нее есть свои ограничения.

В главе 9 представлены разные подходы миграции базы данных, как простые, так и сложные, и обсуждаются плюсы и минусы каждого подхода.

5.9 Использование функции миграции в EF Core для изменения структуры базы данных

В этом разделе описывается, как использовать миграции для обновления базы данных.

Можно использовать миграции и на компьютере, где ведется разработка, и у себя на хосте, но, как объяснено в разделе 5.8.2, наиболее

сложный вариант – это база данных на веб-хосте. В этой книге есть целая глава (глава 9), посвященная миграциям, а в данном разделе представлен обзор использования миграций в приложениях ASP.NET Core.

5.9.1 Обновление рабочей базы данных

Как вы, возможно, помните из главы 2, в которой кратко представлены миграции, в консоли диспетчера пакетов Visual Studio можно набрать две команды:

- `Add-Migration` – создает код миграции в вашем приложении для создания или обновления структуры базы данных;
- `Update-Database` – применяет код миграции к базе данных, на которую ссылается DbContext.

С первой командой все в порядке, но вторая команда обновит только базу данных по умолчанию, которая, скорее всего, будет находиться на компьютере, где ведется разработка. Это не рабочая база. Что происходит, когда вы хотите развернуть свое веб-приложение на каком-то веб-хосте, а база данных находится в состоянии, не соответствующем коду? При использовании миграций есть четыре способа обновить рабочую базу данных:

- можно попросить приложение проверить и обновить базу данных во время запуска;
- можно применить миграцию к базе данных в конвейере непрерывной интеграции (continuous integration, CI) и непрерывной доставки (continuous delivery, CD);
- у вас может быть отдельное приложение для миграции базы данных;
- можно извлечь команды SQL, необходимые для обновления базы данных, а затем использовать какой-то инструмент, чтобы применить эти команды к своей рабочей базе данных.

Самый простой вариант – первый, который я здесь и опишу. У него есть ограничения, например он не предназначен для работы с несколькими экземплярами веб-хостинга (в Azure это называется *горизонтальным масштабированием – scaling out*). Но заставить приложение выполнить миграцию просто, и это хороший отправной шаг в использовании миграции в приложении ASP.NET Core.

> **ПРЕДУПРЕЖДЕНИЕ** Microsoft все же рекомендует обновлять рабочую базу данных с помощью команд SQL, что является наиболее надежным способом. Но для этого требуется выполнить довольно много шагов и использовать инструменты, которых может не быть у вас под рукой, поэтому я расскажу о самом простом подходе с `Database.Migrate`. В главе 9 рассматриваются все аспекты миграций баз данных, включая преимущества и ограничения каждого подхода.

5.9.2 Заставляем приложение обновить базу данных при запуске

Главным преимуществом того, что ваше приложение применяет любые незавершенные миграции базы данных при запуске, является то, что вы не забудете сделать это: развертывание нового приложения остановит старое приложение, а затем запустит новое. Добавив код, выполняющийся при старте приложения, можно вызвать метод `context.Database.Migrate`, который применит все отсутствующие миграции к базе данных до запуска основного приложения – все просто, пока не произойдет сбой, поэтому в главе 9, посвященной миграциям баз данных, обсуждаются все эти вопросы. Но пока остановимся на простом подходе.

Решив применить миграцию при запуске, нужно решить, где вызывать код миграции. Рекомендуемый подход к добавлению любого кода запуска в приложение ASP.NET Core состоит в том, чтобы добавить его в конец метода `Main` в классе `Program` приложения ASP.NET Core. Обычный код из метода `Main` показан в этом фрагменте кода:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
```

Лучший способ добавить код миграции – создать метод расширения, содержащий код EF Core, который вы хотите запустить, и добавить его после вызова `CreateHostBuilder(args).Build()`. В следующем листинге показан класс ASP.NET Core `Program` с одной новой строкой (выделенной жирным шрифтом), добавленной для вызова нашего метода расширения `MigrateDatabaseAsync`.

ПРИМЕЧАНИЕ В этом разделе я буду использовать команды async/await. Подробнее о них рассказывается в разделе 5.10.

Листинг 5.12 Класс ASP.NET Core `Program`, включая метод для миграции базы данных

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();
        await host.MigrateDatabaseAsync();
        await host.RunAsync();
    }
    //… Остальной код не показан;
}
```

Вы изменяете метод Main, чтобы он стал асинхронным и можно было использовать команды async/await в методе MigrateDatabaseAsync

Вызывает метод расширения для миграции базы данных

Этот вызов выполнит метод Startup.Configure, который настраивает сервисы внедрения зависимостей, необходимые для настройки и миграции базы данных

В конце запускается приложение ASP.NET Core

Метод `MigrateDatabaseAsync` должен содержать весь код, который вы хотите запустить при запуске для миграции и, возможно, заполнения базы данных. В следующем листинге показан один из примеров того, как использовать этот метод для миграции.

Листинг 5.13 Метод расширения `MigrateDatabaseAsync` для миграции базы данных

Создает метод расширения, который принимает IHost

Создает поставщика сервисов с областью действия Scoped. После выхода из блока using все сервисы будут недоступны. Этот подход является рекомендуемым способом получения сервисов за пределами HTTP-запроса

```
public static async Task MigrateDatabaseAsync
    (this IHost webHost)
{
    using (var scope = webHost.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        using (var context = services
            .GetRequiredService<EfCoreContext>())
        {
            try
            {
                await context.Database.MigrateAsync();
                // Помещаем сюда любое сложное заполнение базы данных
            }
            catch (Exception ex)
            {
                var logger = services
                    .GetRequiredService<ILogger<Program>>();
                logger.LogError(ex,
                "An error occurred while migrating the database.");

                throw;
            }
        }
    }
}
```

Создает экземпляр DbContext, жизненный цикл которого составляет только внешний оператор using

Вызывает команду EF Core, MigrateAsync, чтобы применить все незавершенные миграции при запуске

При необходимости можно добавить сюда метод для обработки сложного заполнения базы данных

Если возникает исключение, вы регистрируете информацию, чтобы можно было ее диагностировать

Выдает исключение повторно, потому что вы не хотите, чтобы приложение продолжало работу, если возникает проблема с миграцией базы данных

Последовательность вызовов в начале листинга – это рекомендуемый способ получения копии DbContext внутри метода `Configure` в классе ASP.NET Core `Startup`. Этот код создает экземпляр DbContext с жизненным циклом типа scoped (см. раздел 5.3.3), который можно безопасно использовать для доступа к базе данных.

Ключевые команды в листинге 5.13 внутри блока `try` (выделены жирным шрифтом) вызывают команду EF Core `MigrateAsync`. Она применяет любую существующую миграцию, которая еще не была применена к базе данных.

EF6 Подход EF Core к настройке базы данных отличается от подхода, используемого в EF6.x.

При первом использовании DbContext EF6.x выполняет различные проверки, применяя *инициализаторы базы данных*, тогда как EF Core вообще ничего не делает с базой данных при инициализации. Следовательно, для обработки миграций нужно добавить собственный код. Обратная сторона состоит в том, что нужно писать код, но есть и положительный момент: вы можете полностью контролировать происходящее.

НАСТРОЙКА НАЧАЛЬНОГО СОДЕРЖИМОГО БАЗЫ ПРИ ЗАПУСКЕ

Помимо миграции базы данных, можно одновременно добавлять в нее данные по умолчанию, особенно если она пустая. Этот процесс, называемый *заполнением* базы данных, включает в себя добавление исходных данных в базу данных или, возможно, обновление данных в существующей базе. Основной способ заполнить базу статическими данными – это миграция, о которой я расскажу в главе 9. Другой вариант – запустить некий код после завершения миграции.

Это полезно, если у вас есть динамические данные или сложные обновления, с которыми не может справиться заполнение.

Примером выполнения кода после миграции является добавление примеров книг с авторами, отзывами и т. д. в приложение Book App, если там еще нет книг. Для этого создается метод расширения `SeedDatabaseAsync`, показанный в следующем листинге. Код добавляется после вызова метода `Database.MigrateAsync` из листинга 5.13.

Листинг 5.14 Наш метод расширения `MigrateAndSeed`

```
public static async Task SeedDatabaseAsync
    (this EfCoreContext context)
{
    if (context.Books.Any()) return;

    context.Books.AddRange(
        EfTestData.CreateFourBooks());
    await context.SaveChangesAsync();
}
```

Метод расширения, принимающий DbContext

Если есть уже существующие книги, выходим из метода, так как добавлять их не нужно

В базе данных нет книг, поэтому вы заполняете ее данными; в этом случае вы добавляете книги по умолчанию

Вызывается метод SaveChangesAsync для обновления базы данных

Используя метод `SeedDatabaseAsync`, вы проверяете, есть ли в базе данных какие-либо книги, а затем добавляете их, только если база данных пуста (например, только что создана). Это простой пример, а вот и другие:

- загрузка данных из файла при запуске (см. класс `SetupHelpers` в сервисном слое в связанном репозитории GitHub);
- заполнение дополнительных данных после определенной миграции – например, если вы добавили свойство или столбец `FullName` и хотите заполнить его из столбцов `FirstName` и `LastName`.

ПРЕДУПРЕЖДЕНИЕ Я пробовал обновить большую базу данных, как в предыдущем примере с `FullName` с десятками тысяч строк, и мне это не удалось. Ошибка произошла из-за того, что обновление было выполнено через EF Core, и для запуска приложения ASP.NET Core потребовалось так много времени, что в Azure истекло время ожидания веб-приложения. Теперь я знаю, что мне следовало выполнить обновление с помощью SQL в миграции (см. пример в разделе 9.5.2), что было бы намного быстрее.

Если вы хотите запустить свой метод для заполнения базы данных только после применения новой миграции, то можно использовать метод `DbContext Database.GetPendingMigrations`, чтобы получить список миграций, которые должны быть применены. Если этот метод возвращает пустую коллекцию, в текущей базе данных нет незавершенной миграции. Вызов метода `GetPendingMigrations` нужно делать до вызова `Database.Migrate`, поскольку коллекция незавершенных миграций будет пуста после завершения работы метода `Migrate`.

EF6 В EF6.x команда `AddMigration` добавляет класс `Configuration`, содержащий метод под названием `Seed`, который запускается каждый раз при запуске приложения. EF Core использует метод конфигурации `HasData`, позволяющий определять данные, добавляемые во время миграции (глава 9).

5.10 Использование async/await для лучшей масштабируемости

Async/await – это средство, позволяющее разработчику с легкостью использовать *асинхронное программирование,* запуская задачи параллельно. До этого момента в данной книге я не использовал async/await, потому что не объяснил, что это такое. Но нужно знать, что в реальных приложениях, где одновременно выполняется несколько запросов, например ASP.NET Core, большинство команд базы данных будет использовать async/await.

Async/await – обширная тема, но в этом разделе вы узнаете только, как использование данного паттерна может улучшить масштабируемость приложения ASP.NET Core. Он делает это, высвобождая ресурсы, ожидая, пока сервер базы данных выполнит команду (команды), которую(ые) EF Core попросил его выполнить.

ПРИМЕЧАНИЕ Если хотите узнать больше о других функциях async/await, таких как параллельный запуск задач, ознакомьтесь с документацией Microsoft на странице http://mng.bz/nM7K.

5.10.1 Чем паттерн async/await полезен в веб-приложении, использующем EF Core

Когда EF Core обращается к базе данных, ему необходимо дождаться, пока сервер базы данных запустит команды и вернет результат. В случае с большими наборами данных и/или сложными запросами этот процесс может занять сотни миллисекунд или даже секунд. За это время веб-приложение удерживает поток из пула потоков приложения. Каждому подключению к веб-приложению нужен отдельный поток из пула потоков, и существует верхний предел.

Использование версии команды EF Core с async/await означает, что текущий пользовательский поток высвобождается до тех пор, пока не завершится доступ к базе данных, поэтому кто-то другой может использовать его. На рис. 5.8 показаны два случая. В случае А два пользователя одновременно получают доступ к сайту, используя обычный синхронный доступ, и они конфликтуют, поэтому из пула потоков требуется два потока. В случае В, если говорить о пользователе 1, речь идет о доступе к базе данных с длительным временем выполнения запроса, где используется команда async для высвобождения потока, пока он ожидает базу данных.

Это позволяет пользователю 2 повторно использовать поток, высвобожденный командой async, пока пользователь 2 ожидает базу данных.

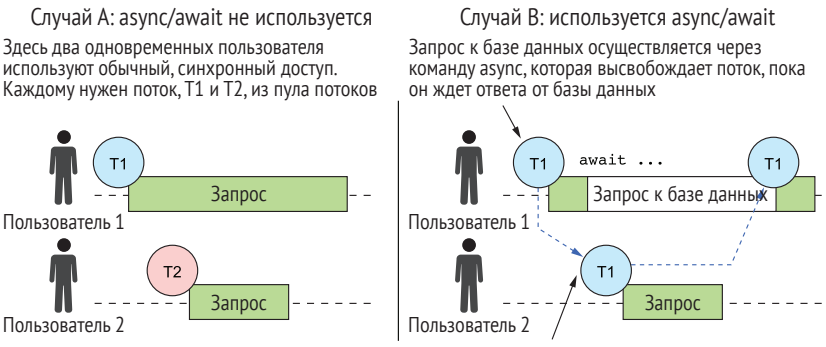


Рис. 5.8 Различия в доступе к базе данных. При обычном синхронном доступе к базе данных в случае А для двух пользователей необходимы два потока. В случае В доступ к базе данных со стороны пользователя 1 выполняется с помощью команды async, которая высвобождает поток Т1, делая его доступным для пользователя 2

> **ПРИМЕЧАНИЕ** Более подробное объяснение того, что делает паттерн async/await в веб-приложении ASP.NET, можно прочитать на странице http://mng.bz/vz7M.

Использование async/await улучшает масштабируемость сайта: веб-сервер сможет обслуживать больше пользователей одновремен-

но. Обратная сторона состоит в том, что выполнение команд async/await занимает немного больше времени, потому что выполняется больше кода. Требуется небольшой анализ, чтобы получить правильный баланс масштабируемости и производительности.

5.10.2 Где использовать async/await для доступа к базе данных?

Microsoft дает универсальный совет – используйте асинхронные методы в веб-приложении везде, где это возможно, потому что они обеспечивают лучшую масштабируемость. Именно это я и делаю в реальных приложениях. Я не делал этого в версиях приложения Book App в первой и второй частях книги только потому, что понимать код без операторов `await` немного проще, но в части III, которая была значительно улучшена, везде используется асинхронный режим.

Синхронные команды немного быстрее, чем эквивалентная команда с использованием async (см. табл. 14.5, где приводятся реальные различия), но разница во времени настолько мала, что соблюдение правила Microsoft «Всегда используйте асинхронные команды в приложениях ASP.NET» – это правильный выбор.

5.10.3 Переход на версии команд EF Core с async/await

Позвольте начать с демонстрации метода, вызывающего асинхронную версию команды EF Core; позже я объясню ее. На рис. 5.9 показан асинхронный метод, возвращающий общее количество книг в базе.

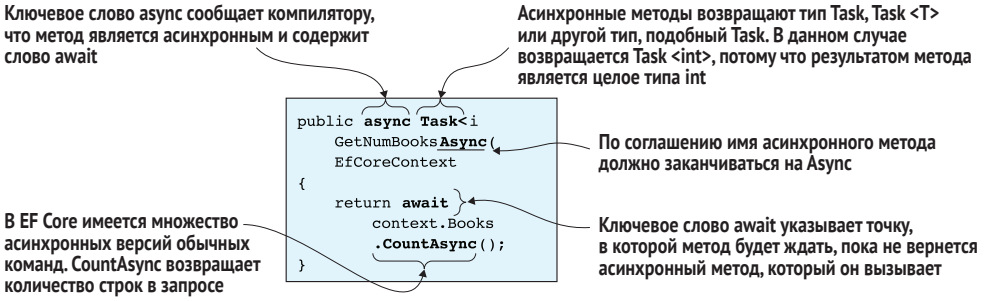


Рис. 5.9 Анатомия асинхронного метода с выделением частей кода, которые отличаются от обычного синхронного метода

EF Core содержит асинхронную версию всех применимых команд, у каждой из которых есть имя метода, оканчивающееся на `Async`. Как было показано в предыдущем примере с асинхронным методом, необходимо придать «асинхронность» методу, где вы вызываете асинхронную команду EF Core.

Правило состоит в том, что после использования этой команды любой вызывающий код должен либо быть асинхронным методом, либо возвращать задачу (Task) напрямую, пока она не дойдет до вызывающего кода верхнего уровня, который должен обработать ее асинхронно. ASP.NET Core поддерживает асинхронность для всех основных команд, таких как методы действия контроллера, поэтому эта ситуация не является проблемой в таком приложении.

В следующем листинге показана асинхронная версия нашего метода действия `Index` из `HomeController` с частями, которые необходимо изменить, чтобы эта команда использовала асинхронный доступ к базе данных. Асинхронные части выделены жирным шрифтом.

Листинг 5.15 Асинхронный метод действия `Index` из `HomeController`

```
public async Task<IActionResult> Index
    (SortFilterPageOptions options)
{
    var listService =
        new ListBooksService(_context);
    var bookList = await listService
        .SortFilterPage(options)
        .ToListAsync();

    return View(new BookListCombinedDto
        (options, bookList));
}
```

Метод действия Index делается асинхронным с помощью ключевого слова async, а возвращаемый тип должен быть обернут в обобщенную задачу

Нужно дождаться результата метода ToListAsync, который является асинхронной командой

Можно изменить метод SortFilterPage на асинхронный, заменив .ToList() на .ToListAsync()

Поскольку мы проектируем наш метод `SortFilterPage` таким образом, чтобы он возвращал `IQueryable<T>`, изменить доступ к базе данных на асинхронный несложно: замените метод `ToList` на `ToListAsync`.

> **СОВЕТ** Код бизнес-логики часто является хорошим кандидатом для использования асинхронных методов доступа к базам данных, потому что их доступ к базе данных нередко содержит сложные команды для чтения и записи данных. Я создал асинхронные версии `BizRunner` на случай, если они вам понадобятся. Вы можете найти их в сервисном слое в каталоге BizRunners (см. http://mng.bz/PPlw).

Еще одна часть асинхронного кода – это `CancellationToken`, механизм, позволяющий останавливать асинхронный метод вручную или по тайм-ауту. Все асинхронные команды LINQ и EF Core, например `SaveChangesAsync`, принимают необязательный `CancellationToken`. В разделе 5.11 показано использование `CancellationToken` для остановки любых повторяющихся фоновых задач при остановке ASP.NET Core.

5.11 Выполнение параллельных задач: как предоставить DbContext

В некоторых ситуациях полезно запускать более одного потока кода. Под этим я подразумеваю выполнение отдельной *задачи* – параллельного набора кода, который выполняется «одновременно» с основным приложением. Я заключил слово «одновременно» в кавычки, потому что если процессор только один, две задачи должны использовать его совместно.

Параллельные задачи полезны в различных случаях. Допустим, вы получаете доступ к нескольким внешним источникам, и нужно подождать, прежде чем они вернут результат. Используя несколько задач, выполняемых параллельно, вы улучшаете производительность. В другом сценарии у вас может быть длительная задача, например обработка выполнения заказа в фоновом режиме. Вы используете параллельные задачи, чтобы не блокировать обычный поток и не делать сайт медленным и невосприимчивым. На рис. 5.10 показан пример фоновой задачи, в которой процесс с длительным временем выполнения запускается в другом потоке, чтобы не задерживать пользователя.

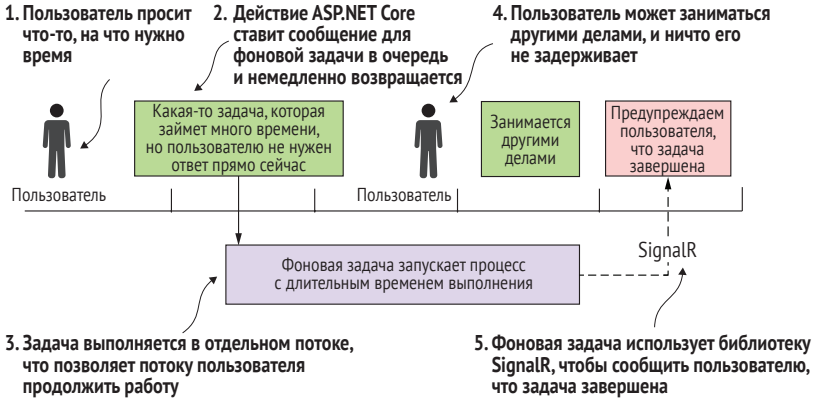


Рис. 5.10 Перенос процессов с длительным временем выполнения в фоновую задачу, которая выполняется параллельно с основным сайтом, что делает его более отзывчивым. В этом примере я использую класс ASP.NET Core `backgroundService` для запуска задачи с длительным временем выполнения. По завершении задачи применяется `SignalR` для обновления экрана пользователя с сообщением о том, что длительная задача успешно завершена. (`SignalR` – это библиотека, позволяющая приложению ASP.NET Core отправлять сообщения подключенному клиенту)

Запуск параллельных задач относится не только к ASP.NET Core; это может произойти в любом приложении. Но крупные веб-приложения часто используют эту функцию, поэтому я объясню ее в данной главе.

Решение, которое мы создадим, представляет собой фоновую службу, которая запускается каждый час и регистрирует количество рецензий на книги в базе данных. Этот простой пример покажет вам, как сделать две вещи:

- получить экземпляр DbContext для параллельного запуска;
- использовать `IHostedService` для запуска фоновой задачи.

5.11.1 Получение экземпляра DbContext для параллельного запуска

Если вы хотите выполнять код, использующий EF Core параллельно, то не можете применять обычный подход к получению DbContext приложения, потому что он не является потокобезопасным; нельзя использовать один и тот же экземпляр в нескольких потоках. EF Core выбросит исключение, если обнаружится, что один и тот же экземпляр DbContext применяется в двух задачах.

Правильный способ заставить DbContext работать в фоновом режиме – использовать сервис внедрения зависимостей с жизненным циклом типа scoped. Этот сервис позволяет создавать через внедрение зависимостей уникальный DbContext для задачи, которую вы выполняете. Для этого нужно сделать три вещи:

- получить экземпляр `IServiceScopeFactory`, используя внедрение через конструктор;
- использовать `IServiceScopeFactory` для *сервиса внедрения зависимостей с жизненным циклом типа scoped*;
- использовать этот сервис, чтобы получить экземпляр DbContext, который является уникальным.

Листинг 5.16 Метод внутри фоновой задачи, который обращается к базе данных

Использует ScopeProviderFactory для создания нового поставщика внедрения зависимостей с жизненным циклом типа scoped

IHostedService вызовет этот метод по истечении установленного периода

```
private async Task DoWorkAsync(CancellationToken stoppingToken)
{
    using (var scope = _scopeFactory.CreateScope())
    {
        var context = scope.ServiceProvider
            .GetRequiredService<EfCoreContext>();
        var numReviews = await context.Set<Review>()
            .CountAsync(stoppingToken);
        _logger.LogInformation(
            "Number of reviews: {numReviews}", numReviews);
    }
}
```

Подсчитывает отзывы, используя асинхронный метод. Мы передаем stoppingToken методу async, потому что это хорошая практика

Регистрирует информацию

Из-за поставщика внедрения зависимостей с жизненным циклом типа scoped созданный экземпляр DbContext будет отличаться от всех других экземпляров DbContext

Здесь есть важный момент: мы предоставляем `ServiceScopeFactory` каждой задаче, чтобы она могла использовать внедрение зависимостей для получения уникального экземпляра DbContext (и любых других сервисов с жизненным циклом типа scoped). Помимо решения проблемы потокобезопасности DbContext, если вы выполняете метод многократно, лучше всего иметь новый экземпляр DbContext, чтобы данные последнего запуска не повлияли на следующий запуск.

5.11.2 Запуск фоновой службы в ASP.NET Core

Ранее я описал, как получить потокобезопасную версию DbContext; теперь вы будете использовать ее в фоновой задаче. Следующий пример не такой сложный, как показано на рис. 5.10, но он описывает, как писать и запускать фоновые задачи.

В ASP.NET Core есть возможность запускать задачи в фоновом режиме. На самом деле эта ситуация не является проблемой базы данных, но для полноты я покажу вам код. (Рекомендую просмотреть справочную документацию ASP.NET Core по фоновым задачам на странице http://mng.bz/QmOj.) В этом листинге показан код, который выполняется в другом потоке и вызывает метод `DoWorkAsync`, показанный в листинге 5.16, каждый час.

Листинг 5.17 Фоновая служба ASP.NET Core, вызывающая метод `DoWorkAsync` каждый час

IServiceScopeFactory внедряется сервисом внедрения зависимостей и используется для создания новой области

Наследование класса BackgroundService означает, что этот класс может непрерывно работать в фоновом режиме

```
public class BackgroundServiceCountReviews : BackgroundService
{
    private static TimeSpan _period =
        new TimeSpan(0,1,0,0);

    private readonly IServiceScopeFactory _scopeFactory;
    private readonly ILogger<BackgroundServiceCountReviews> _logger;

    public BackgroundServiceCountReviews(
        IServiceScopeFactory scopeFactory,
        ILogger<BackgroundServiceCountReviews> logger)
    {
        _scopeFactory = scopeFactory;
        _logger = logger;
    }

    protected override async Task ExecuteAsync
        (CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await DoWorkAsync(stoppingToken);
```

Устанавливает задержку между каждым вызовом кода для регистрации количества отзывов

У класса BackgroundService есть метод ExecuteAsync, который мы переопределяем, чтобы добавить собственный код

Этот цикл многократно вызывает метод DoWorkAsync, после чего выполняет задержку

```
            await Task.Delay(_period, stoppingToken);
        }
    }

    private async Task DoWorkAsync…
    //Смотрите листинг 5.16;
}
```

Необходимо зарегистрировать свою фоновую задачу в поставщике внедрения зависимостей с помощью метода `AddHostedService`. Когда приложение Book App запустится, ваша фоновая задача будет запущена первой, но когда она попадет туда, где вызывается асинхронный метод и используется оператор `await`, управление возвращается к коду ASP.NET Core, который запускает веб-приложение.

5.11.3 Другие способы получения нового экземпляра DbContext

Хотя внедрение зависимостей – это рекомендуемый способ получения DbContext, в некоторых случаях, например в консольном приложении, оно может быть не настроено или недоступно. Тогда есть два других варианта, позволяющих получить экземпляр DbContext:

- переместите конфигурацию DbContext, переопределив метод `OnConfiguring` в DbContext и разместив код настройки DbContext там;
- используйте тот же конструктор, что и для ASP.NET Core, и вручную внедрите параметры базы данных и строку подключения, как в модульных тестах (см. главу 17).

Обратная сторона первого варианта состоит в том, что в нем используется фиксированная строка подключения, поэтому он всегда обращается к одной и той же базе данных, а это может затруднить развертывание в другой системе при изменении имени базы данных или параметров. Второй вариант – предоставление параметров базы данных вручную – позволяет читать строку подключения из файла appsettings.json или получать ее другим путем.

Еще одна проблема, о которой следует помнить, заключается в том, что каждый вызов дает вам новый экземпляр DbContext. Судя по обсуждению типов жизненных циклов в разделе 5.3.3, иногда может потребоваться один и тот же экземпляр DbContext, чтобы гарантировать, что отслеживание изменений работает. Можно обойти эту проблему, разработав свое приложение таким образом, чтобы один экземпляр DbContext передавался по всему коду, который необходим для совместной работы над обновлениями базы данных.

Резюме

- ASP.NET Core использует внедрение зависимостей для предоставления DbContext. С его помощью можно динамически связывать

части приложения, позволяя внедрению зависимостей создавать экземпляры классов по мере необходимости.

- Метод `ConfigureServices` в классе ASP.NET Core `Startup` – это место для настройки и регистрации версии DbContext с помощью строки подключения, которая помещается в файл настроек приложения ASP.NET Core.
- Чтобы получить экземпляр DbContext, который будет использоваться с вашим кодом через внедрение зависимостей, можно использовать внедрение через конструктор. Внедрение зависимостей будет смотреть на тип каждого параметра конструктора и пытаться найти сервис, который сможет предоставить экземпляр.
- Код доступа к базе данных можно создать как сервис и зарегистрировать. Затем можно внедрить свои сервисы в методы действий ASP.NET Core с помощью внедрения через параметр: внедрение зависимостей найдет сервис, который предоставит значение для параметра метода действия ASP.NET Core, помеченного атрибутом `[FromServices]`.
- Для развертывания приложения ASP.NET Core, использующего базу данных, необходимо определить строку подключения к базе данных, в которой указано расположение и имя базы на хосте.
- Миграции предоставляют один из способов изменения базы данных при изменении классов сущностей и/или конфигурации EF Core. У метода `Migrate` есть некоторые ограничения при использовании на сайтах облачного хостинга, где выполняется несколько экземпляров вашего веб-приложения.
- Методы async/await для кода доступа к базе данных могут заставить ваш сайт обслуживать больше одновременных пользователей, но производительность может пострадать, особенно при простом доступе к базе данных.
- Если вы хотите использовать параллельные задачи, необходимо предоставить уникальный экземпляр DbContext, создав нового поставщика внедрения зависимостей с жизненным циклом типа scoped.

Для читателей, знакомых с EF6.x:

- способ получить экземпляр DbContext в ASP.NET Core – использовать внедрение зависимостей;
- по сравнению с EF6.x, EF Core использует другой подход к созданию первого экземпляра DbContext. В EF6.x есть инициализаторы базы данных, и он может выполнять метод `Seed`. В EF Core нет ни одной из этих функций EF6.x, но он оставляет вам возможность написать конкретный код, который вы хотите выполнить при запуске;
- заполнение базы данных в EF Core отличается от того, как это делается в EF6.x. EF Core добавляет заполнение начальными значениями в миграции, поэтому оно выполняется только в том случае, если миграция применяется к базе данных; см. главу 9 для получения дополнительной информации.

6 Советы и техники, касающиеся чтения и записи данных с EF Core

В этой главе рассматриваются следующие темы:

- выбор правильного подхода к чтению данных из базы данных;
- написание запросов, которые хорошо работают на стороне базы данных;
- избежание проблем при использовании фильтров запросов и специальных команд LINQ;
- использование AutoMapper для более быстрого написания запросов выборки данных;
- написание кода для быстрого копирования и удаления сущностей в базе данных.

Первые четыре главы посвящены различным способам чтения и записи данных в базу данных, а в главе 5 мы использовали эти сведения для создания приложения Book App. В этой главе собрано множество различных советов и методов, касающихся чтения и записи данных с EF Core.

Глава поделена на два раздела: чтение из базы данных и запись в базу данных. В каждом разделе рассматриваются определенные

проблемы, с которыми вы можете столкнуться, и в то же время объясняется, как EF Core решает их. Цель состоит в том, чтобы дать вам ряд практических советов через решение различных задач и в то же время возможность углубить свои познания относительно того, как работает EF Core. Это полезные советы, и в долгосрочной перспективе, повышая свою экспертность в EF Core, вы сможете писать более качественный код.

> **СОВЕТ** Не забудьте, что в репозитории Git (http://mng.bz/XdlG) содержатся модульные тесты для каждой главы. Для этой главы в проекте `Test` в ветке master найдите классы, имена которых начинаются с Ch06_. Иногда увидеть код полезнее, чем читать о нем.

6.1 Чтение из базы данных

В этом разделе рассматриваются различные аспекты и примеры чтения данных из базы данных. Цель состоит в том, чтобы познакомить вас с некоторыми особенностями внутренней работы EF Core, рассматривая разные проблемы и вопросы. Попутно вы будете получать советы, которые могут пригодиться, когда вы будете создавать приложения с EF Core. Ниже приводится список тем, касающихся чтения данных из базы данных, используя EF Core:

- изучение этапа ссылочной фиксации в запросе;
- понимание того, что делает метод `AsNoTracking` и его разновидности;
- эффективное чтение иерархических данных;
- понимание того, как работает метод `Include`;
- обеспечение отказоустойчивости загрузки навигационных коллекций;
- использование фильтров запросов в реальных ситуациях;
- рассмотрение команд LINQ, требующих особого внимания;
- использование AutoMapper для автоматизации построения запросов выборки данных;
- оценка того, как EF Core создает класс сущности при чтении данных.

6.1.1 Этап ссылочной фиксации в запросе

Когда вы делаете запрос к базе данных с помощью EF Core, запускается этап, называемый *ссылочной фиксацией* (*relational fixup*), который используется для заполнения навигационных свойств других классов сущностей, включенных в запрос. Я описывал этот процесс в разделе 1.9.2, где сущность `Book` была связана с `Author`. До данного момента все запросы, которые вы видели, связывали только классы сущностей,

считываемые текущим запросом. Но на самом деле ссылочная фиксация в обычном запросе на чтение и запись данных может связываться с любыми отслеживаемыми сущностями за пределами одного запроса, как описано в этом разделе.

Всякий раз, когда вы считываете классы сущностей как отслеживаемые (ваш запрос не включает команду `AsNoTracking`), будет запускаться этап ссылочной фиксации для связывания навигационных свойств. Здесь есть важный момент: на данном этапе просматриваются не только данные в запросе; при заполнении навигационных свойств также просматриваются все существующие отслеживаемые сущности. На рис. 6.1 показаны два способа загрузки сущности `Book` и отзывов на нее. В обоих случаях заполняется навигационное свойство `Reviews`.

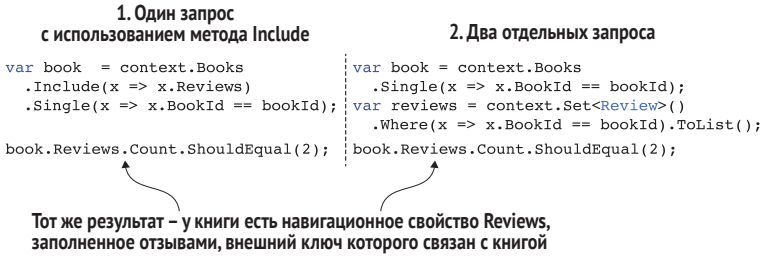


Рис. 6.1 На этом рисунке показан один запрос, который загружает книгу (Book) с отзывами (Review), используя метод `Include` для загрузки отзывов (см. код слева). Запрос справа загружает книгу без отзывов; затем выполняется второй запрос, который загружает отзывы отдельно. Обе версии кода дают одинаковый результат: загружается сущность Book и ее навигационное свойство `Reviews`, причем отзывы связаны с книгой

Как показывает этот простой пример, ссылочная фиксация, которая запускается после завершения запроса, заполняет все навигационные ссылки на основе ограничений по ключу, и это довольно мощная вещь. Например, если загрузить `Books`, `Reviews`, `BookAuthor` и `Authors` отдельными запросами, то EF Core правильно свяжет все навигационные свойства. Следующий фрагмент кода именно это и делает: книги, прочитанные в первой строке, еще не имеют связей с другими объектами, но к четвертой строке навигационные свойства книги `Reviews` и `AuthorsLink` заполняются, как и навигационные свойства `BookAuthor`, `Book` и `Author`:

```
var books = context.Books.ToList();
var reviews = context.Set<Review>().ToList();
var authorsLinks = context.Set<BookAuthor>().ToList();
var authors = context.Authors.ToList();
```

Эта особенность EF Core позволяет делать некоторые полезные вещи. В разделе 6.1.3 вы научитесь эффективно читать иерархические данные с помощью этой техники.

6.1.2 Понимание того, что делает метод AsNoTracking и его разновидности

Когда вы выполняете запрос к базе данных через EF Core, то делаете это по какой-то причине: чтобы изменить прочитанные данные, например изменить свойство `Title` в сущности `Book`, или выполнить запрос с доступом только на чтение, например чтобы отобразить книги с ценами, авторами и т. д. В этом разделе рассказывается, как методы `AsNoTracking` и `AsNoTrackingWithIdentityResolution` повышают производительность запроса с доступом только на чтение и влияют на считываемые данные. Следующий фрагмент кода из главы 1 использует метод `AsNoTracking` для отображения списка книг и их авторов в консоли:

```
var books = context.Books
            .AsNoTracking()
            .Include(a => a.Author)
            .ToList();
```

Обычный запрос без какого-либо из двух методов `AsNoTracking` будет отслеживать классы сущностей, загружаемые запросом, что позволяет обновлять или удалять классы сущностей, которые вы загрузили. Но если вам нужен запрос только для чтения, то можно включить в него один из этих методов. Они повышают производительность и гарантируют, что изменения в данных не будут записываться обратно в базу, но есть небольшие различия в возвращаемых связях:

- метод `AsNoTracking` сокращает время запроса, но не всегда отображает точные связи с объектами базы данных;
- метод `AsNoTrackingWithIdentityResolution`, как правило, быстрее обычного запроса, но медленнее, чем тот же запрос с `AsNoTracking`. Улучшение состоит в том, что связи с объектами базы данных представляются правильно, экземпляры классов сущностей с одинаковым первичным ключом представлены одним объектом, который соответствует одной строке в базе данных.

Начнем с рассмотрения различий в данных, возвращаемых запросом, который использует два варианта метода `AsNoTracking`. Чтобы обеспечить максимальную производительность, метод `AsNoTracking` не выполняет функцию, называемую разрешением идентичности, которая обеспечивает наличие только одного экземпляра сущности для каждой строки в базе данных. Без применения этой функции к запросу вы можете получить дополнительные экземпляры классов сущностей.

На рис. 6.2 показано, что происходит, когда вы используете методы `AsNoTracking` и `AsNoTrackingWithIdentityResolution` в суперпростой базе данных из главы 1. В этом примере четыре книги, но у первых двух книг один и тот же автор. Как показано на рисунке, запрос с `As-`

NoTracking создает четыре экземпляра класса Author, но в базе данных только три строки в таблице Author.

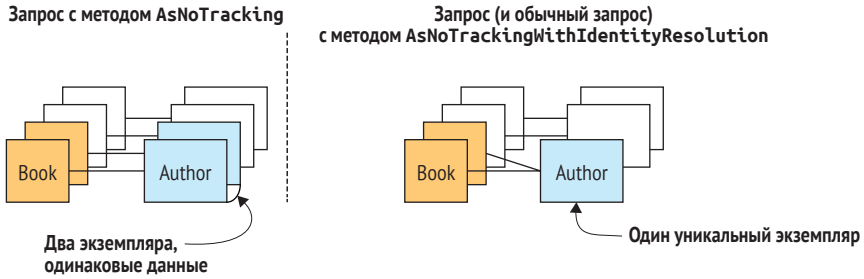


Рис. 6.2 Первые две книги написаны одним и тем же автором, Мартином Фаулером. В запросе с AsNoTracking слева EF Core создает четыре экземпляра класса Author, два из которых содержат одинаковые данные. Запрос с AsNoTrackingWithIdentityResolution (или обычный запрос) справа создает только три экземпляра класса Author, а первые две книги указывают на один и тот же экземпляр

В большинстве запросов на чтение, например при отображении каждой книги с именем ее автора, наличие четырех экземпляров класса Author не имеет значения, поскольку повторяющиеся экземпляры содержат те же данные. В этих типах запросов нужно использовать метод AsNoTracking, потому что он производит самый быстрый запрос.

Но если вы каким-то образом используете связи, например для создания отчета о книгах, которые связаны с другими книгами того же автора, метод AsNoTracking может вызвать проблемы. В таком случае нужно использовать метод AsNoTrackingWithIdentityResolution.

ИСТОРИЧЕСКАЯ СПРАВКА Немного истории: до появления EF Core версии 3.0 метод AsNoTracking включал этап разрешения идентичности, но в EF Core 3.0, где большое внимание уделялось производительности, эта функция была удалена из метода AsNoTracking, что привело к проблемам с существующими приложениями. Поэтому в EF Core 5 для устранения этих проблем был добавлен метод AsNoTrackingWithIdentityResolution.

Чтобы у вас сложилось представление о различиях в производительности, я выполнил простой тест из трех запросов, загрузив сотню книг с сущностями Reviews, BookAuthor и Author.

В табл. 6.1 показан хронометраж (второй запрос).

Как видите, метод AsNoTracking – самый быстрый в этом (ненаучном) тесте и примерно вдвое быстрее обычного запроса, поэтому его стоит использовать. Метод AsNoTrackingWithIdentityResolution лишь немного быстрее (в данном случае), чем обычный запрос с доступом на чтение и запись, но, как и в версии с AsNoTracking, сущности не от-

слеживаются, что улучшает эффективность метода `SaveChanges`, когда он ищет обновленные данные.

Таблица 6.1 Результат выполнения одного и того же запроса с использованием обычного запроса для чтения-записи и запросов, содержащих методы `AsNoTracking` и `AsNoTrackingWithIdentityResolution`

Разновидности метода `AsNoTracking`	Время (мс)	Разница в процентах
Без метода `AsNoTracking` (обычный запрос)	95	100 %
`AsNoTracking`	40	42 %
`AsNoTrackingWithIdentityResolution`	85	90 %

Еще одна особенность методов `AsNoTracking` и `AsNoTrackingWithIdentityResolution` заключается в том, что этап ссылочной фиксации (см. раздел 6.1.1) работает только в рамках запроса. В результате два запроса, использующих `AsNoTracking` или `AsNoTrackingWithIdentityResolution`, создадут новые экземпляры каждой сущности, даже если первый запрос загрузил те же данные. При обычных запросах два отдельных запроса будут возвращать одни и те же экземпляры класса сущности, потому что этап ссылочной фиксации работает для всех отслеживаемых сущностей.

6.1.3 Эффективное чтение иерархических данных

Однажды я работал с клиентом, у которого было много *иерархических данных* – данных, которые имеют ряд связанных классов сущностей с неопределенной глубиной. Проблема заключалась в том, что мне приходилось разбирать всю иерархию, прежде чем я мог ее отобразить. Сначала я делал это, используя немедленную загрузку для первых двух уровней; затем я использовал явную загрузку для более глубоких уровней. Это работало, но производительность была низкой, а база данных была перегружена множеством одиночных запросов.

Данная ситуация заставила меня задуматься: если обычная ссылочная фиксация запросов настолько умная, может ли это помочь мне улучшить производительность запроса? И у меня получилось! Позвольте мне привести пример, где используются сотрудники одной компании. На рис. 6.3 показана иерархическая структура компании, которую мы хотим загрузить.

Можно использовать `.Include(x => x.WorksForMe).ThenInclude(x => x.WorksForMe)` и т. д., но одного `.Include(x => x.WorksForMe)` достаточно, поскольку ссылочная фиксация сможет обработать остальное. В следующем листинге представлен пример, в котором вам нужен список всех сотрудников, занимающихся разработкой, и их связи. LINQ в этом запросе транслируется в один SQL-запрос.

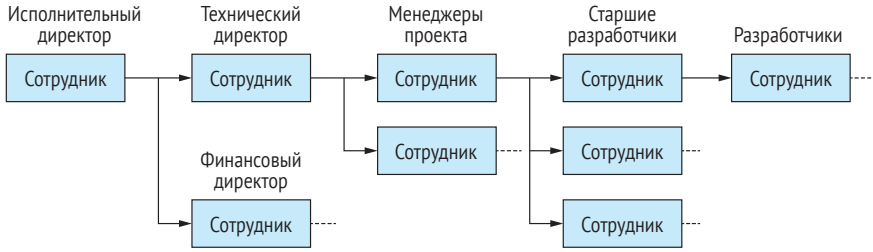


Рис. 6.3 Один из примеров иерархических данных. Проблема с данными такого рода состоит в том, что вы не знаете, насколько они глубокие. Но, оказывается, все, что нам нужно, – это один `.Include(x => x.WorksForMe)`. Тогда этап ссылочной фиксации правильно свяжет иерархические данные

Листинг 6.1 Загрузка всех сотрудников, занимающихся разработкой, и их связей

```
var devDept = context.Employees
    .Include(x => x.WorksForMe)
    .Where(x => x.WhatTheyDo.HasFlag(Roles.Development))
    .ToList();
```

В базе данных содержатся все сотрудники

Один вызов метода Include – все, что вам нужно; ссылочная фиксация выяснит, что с чем связано

Отфильтровывает сотрудников, чтобы найти тех, кто занимается разработкой

В листинге 6.1 представлена отслеживаемая версия иерархических данных, но если вам нужна версия только для чтения, то можно добавить в запрос метод `AsNoTrackingWithIdentityResolution`. Обратите внимание, что метод `AsNoTracking` не будет работать, потому что привязка связей зависит от функции ссылочной фиксации EF Core, которая отключена в методе `AsNoTracking`.

Прежде чем найти этот подход, я использовал явную загрузку, что приводило к неэффективным запросам. Перейдя на данный подход, я сократил время выполнения одного запроса, а также снизил нагрузку на сервер базы данных.

ПРИМЕЧАНИЕ Нужно решить, для каких связей использовать метод `Include`. В данном случае у меня есть навигационное свойство `Manager` (одиночное) и навигационное свойство `WorksForMe` (коллекция). Оказывается, включение свойства `WorksForMe` заполняет и коллекцию `WorksForMe`, и свойство `Manager`. Но включение навигационного свойства `Manager` означает, что коллекция `WorksForMe` создается только при наличии сущностей для привязки, в противном случае коллекция `WorksForMe` будет пустой. Я не знаю, почему два этих метода `Include` разные, поэтому я все тестирую – чтобы убедиться, что знаю, как работает EF Core.

6.1.4 Понимание того, как работает метод Include

Самый простой способ загрузить класс сущности и ее связи – использовать метод `Include`, который прост в применении и обычно обеспечивает эффективный доступ к базе данных. Но стоит знать, как он работает и на что следует обратить внимание.

С появлением EF Core 3.0 способ преобразования метода `Include` в SQL изменился. Изменение обеспечило повышение производительности во многих ситуациях, но в случае с некоторыми сложными запросами это отрицательно сказалось на производительности.

Возьмем пример из базы данных приложения Book App и посмотрим, как загружается `Book` с отзывами и авторами. В следующем фрагменте кода показан запрос:

```
var query = context.Books
    .Include(x => x.Reviews)
    .Include(x => x.AuthorsLink)
         .ThenInclude(x => x.Author);
```

На рис. 6.4 показаны различные SQL-запросы, создаваемые EF Core 2.2 и EF Core 3.0 для книги, у которой четыре отзыва и два автора.

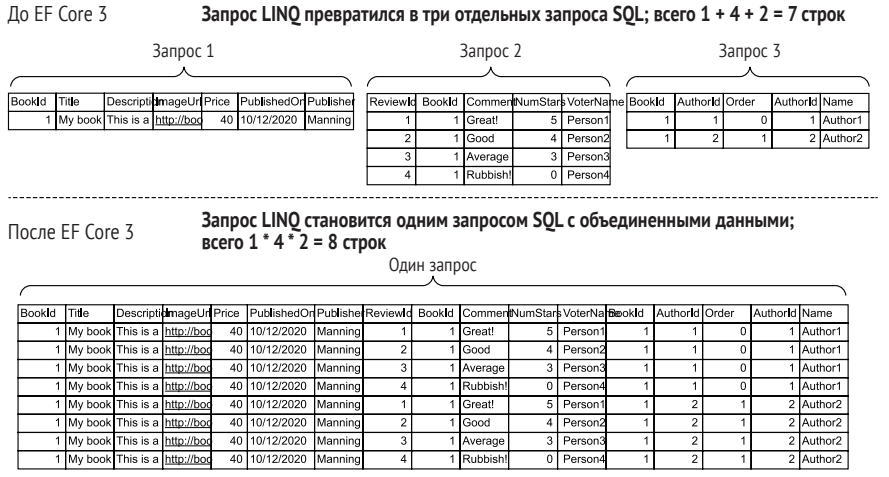


Рис. 6.4 Сравнение способа загрузки данных в EF Core до и после выпуска EF Core 3. Версия сверху – это то, как EF Core работал до версии 3: он использовал отдельные запросы к базе данных для чтения любых коллекций. Версия внизу – это то, что делает EF Core 3 и более поздние версии: он объединяет все данные в один большой запрос

Преимущество способа обработки загрузки связанных коллекций в EF Core 3.0 заключается в производительности, которая во многих ситуациях оказывается выше. Я провел простой эксперимент, загрузив книги с десятью отзывами и двумя авторами в EF Core 2.1 и EF Core 3.0.

EF Core 3.0 был примерно на 20 % быстрее. Но в некоторых конкретных ситуациях он действительно может быть очень медленным, о чем я расскажу ниже.

Проблемы с производительностью возникают, если у вас несколько связанных коллекций, которые вы хотите включить в запрос, и у некоторых из этих связей большое количество записей в коллекции. Можно увидеть эту проблему, посмотрев на вычисления в правой части рис. 6.4. Здесь показано, что количество строк, прочитанных с использованием версии EF Core до 3.0, вычисляется путем *сложения* строк. Но в EF Core 3.0 и более поздних версиях количество прочитанных строк рассчитывается путем *умножения*. Предположим, вы загружаете 3 связи, в каждой из которых 100 строк. Версия EF Core до 3.0 считывала 100 + 100 + 100 = 300 строк, но EF Core 3.0 и более поздние версии прочитают 100 * 100 * 100 = 1 миллион строк.

Чтобы увидеть проблемы с производительностью, я создал тест, в котором у сущности было три связи «один ко многим», и у каждой из них было 100 строк в базе данных. В следующем фрагменте показан обычный подход к загрузке связей в запросе с методом `Include`, на который ушло 3500 миллисекунд (ужасный результат!).

```
var result = context.ManyTops
    .Include(x => x.Collection1)
    .Include(x => x.Collection2)
    .Include(x => x.Collection3)
    .Single(x => x.Id == id);
```

К счастью, EF Core 5 предоставляет метод `AsSplitQuery`, который указывает EF Core читать каждый метод `Include` отдельно, как показано в следующем листинге. На эту операцию ушло всего 100 миллисекунд, что примерно в 50 раз быстрее.

Листинг 6.2 Чтение связей по отдельности и возможность объединения их с помощью ссылочной фиксации

```
var result = context.ManyTops
    .AsSplitQuery()
    .Include(x => x.Collection1)
    .Include(x => x.Collection2)
    .Include(x => x.Collection3)
    .Single(x => x.Id == id)
```

Заставляет загружать каждый метод Include отдельно, тем самым решая проблему умножения числа прочитанных строк

Если вы обнаружите, что запрос, в котором используется несколько методов `Include`, выполняется медленно, возможно, это связано с тем, что две или более включенных коллекций содержат много записей. В таком случае добавьте перед этими методами метод `AsSplitQuery`, чтобы переключиться на раздельную загрузку каждой включенной коллекции.

6.1.5 Обеспечение отказоустойчивости загрузки навигационных коллекций

Я всегда стараюсь делать любой код отказоустойчивым. Под этим я подразумеваю, что если я сделаю ошибку в коде, то предпочитаю, чтобы было выброшено исключение, вместо того чтобы молча совершать некорректные действия. Один из моментов, который меня волнует, – это то, как не забыть добавить правильный набор методов `Include` при загрузке сущности со связями. Кажется, я никогда не забываю об этом, но в приложениях с большим количеством связей такое может легко случиться. На самом деле я делал это много раз, в том числе в приложениях моих клиентов, поэтому использую отказоустойчивый подход. Позвольте мне объяснить проблему и предложить ее решение.

Что касается любого навигационного свойства, использующего коллекцию, то я часто вижу, как разработчики назначают пустую коллекцию навигационному свойству в конструкторе либо используя присвоение свойству (см. следующий листинг).

Листинг 6.3 Класс сущности с навигационными коллекциями, заданный как пустая коллекция

```
public class BookNotSafe
{
    public int Id { get; set; }
    public ICollection<ReviewNotSafe> Reviews { get; set; }

    public BookNotSafe()
    {
        Reviews = new List<ReviewNotSafe>();
    }
```

У навигационного свойства Reviews много записей, то есть связь «один ко многим»

Навигационное свойство Reviews инициализируется пустой коллекцией, что упрощает добавление ReviewNotSave к навигационному свойству при создании первичной сущности BookNotSafe

Разработчики делают это, чтобы упростить добавление записей в навигационную коллекцию для вновь созданного экземпляра класса сущности. Но есть и обратная сторона. Если вы забудете вызвать метод `Include` для загрузки коллекции навигационных свойств, то получите пустую коллекцию, тогда как в базе могут быть данные, которые должны заполнить эту коллекцию.

Есть еще одна проблема, если вы хотите заменить всю коллекцию. При отсутствии вызова `Include` старые записи из базы данных не удаляются, поэтому вы получаете комбинацию новых и старых сущностей, а это не то, что вам нужно. В следующем фрагменте кода (адаптировано из листинга 3.17) вместо замены двух существующих отзывов в базе данных оказывается три отзыва:

```
var book = context.Books
      //Отсутствует .Include(x => x.Reviews)
```

```
        .Single(p => p.BookId == twoReviewBookId);

book.Reviews = new List<Review>{ new Review{ NumStars = 1}};
context.SaveChanges();
```

Еще одна веская причина не инициализировать навигационное свойство пустой коллекцией – это производительность.

Например, если вам нужно использовать явную загрузку коллекции и вы знаете, что она уже загружена, потому что она не пустая, то можно пропустить (избыточную) явную загрузку.

Кроме того, в главе 13 я выбираю наиболее эффективный способ добавления нового класса сущности `Review` к классу сущности `Book`, в зависимости от того, загружено ли уже свойство коллекции `Reviews`.

Поэтому в своем коде (и на протяжении всей книги) я не инициализирую никакие навигационные свойства пустой коллекцией. Вместо того чтобы молча пропустить ошибку, когда я пропускаю метод `Include`, я получаю исключение `NullReferenceException`, когда код обращается к навигационному свойству. На мой взгляд, такой результат намного лучше, чем получение неверных данных.

6.1.6 Использование глобальных фильтров запросов в реальных ситуациях

Глобальные фильтры запросов (сокращенно – *фильтры запросов*) были представлены в разделе 3.5 для реализации функции мягкого удаления. В этом разделе вы узнаете о проблемах, связанных с использованием мягкого удаления в реальных приложениях, а также о том, как использовать фильтры запросов для создания мультитенантных систем.

МЯГКОЕ УДАЛЕНИЕ В РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ

Мягкое удаление полезно, потому что пользователи приложения получают второй шанс, когда что-то удаляют. У двух моих клиентов были приложения, где эта функция использовалась почти для каждого класса сущности. Как правило, обычный пользователь что-то удаляет, и это выполняется как мягкое удаление, а администратор может восстановить этот элемент. Оба приложения были сложными и совершенно разными, поэтому я многое узнал о реализации мягкого удаления.

Во-первых, мягкое удаление работает не так, как обычная команда удаления для базы данных. При удалении из базы данных, если вы удалите `Book`, то также удалите все связанные с ней `PriceOffer`, `Reviews` и `AuthorLinks` (см. раздел 3.5.3). Такого не происходит при мягком удалении, у которого есть некоторые интересные моменты.

Если, например, мягко удалить `Book`, то `PriceOffer`, `Reviews` и `AuthorLinks` останутся, а это может вызвать проблемы, если вы не все продумаете. В разделе 5.11.1 мы создали фоновый процесс, который каждый час журналировал количество отзывов в базе данных. Если мягко

удалить книгу, у которой было десять отзывов, можно ожидать, что количество отзывов уменьшится, но с кодом из листинга 5.14 этого не произойдет. Нужен способ справиться с этой проблемой.

В подобной ситуации нам поможет паттерн предметно-ориентированного проектирования (DDD) «Корень и агрегаты». В этом паттерне класс сущности `Book` – это корень, а `PriceOffer`, `Reviews` и `AuthorLinks` – это агрегаты. (См. описание основных и зависимых сущностей в разделе 3.1.1.) Обращаться к агрегатам нужно только через корень. Этот процесс хорошо работает с мягким удалением, потому что если сущность `Book` (корень) мягко удаляется, вы не сможете получить доступ к ее агрегатам. Таким образом, правильный код для подсчета всех отзывов с учетом мягкого удаления выглядит так:

```
var numReviews = context.Books.SelectMany(x => x.Reviews).Count();
```

> **ПРИМЕЧАНИЕ** Еще один способ решить проблему корня и агрегатов с мягким удалением – сымитировать каскадное удаление при настройке мягкого удаления, что довольно сложно сделать. Но я создал библиотеку `EfCore.SoftDeleteServices`, которая имитирует поведение каскадного удаления, но использует мягкое удаление: https://github.com/JonPSmith/EfCore.SoftDeleteServices.

Во-вторых, не следует применять мягкое удаление к связи «один к одному». У вас возникнут проблемы, если вы попытаетесь добавить новую сущность с такой связью при наличии существующей, но мягко удаленной сущности. Если у вас есть мягко удаленная сущность `PriceOffer`, у которой есть связь «один к одному» с `Book`, и вы попытаетесь добавить еще одну `PriceOffer`, то получите исключение базы данных. Связь «один к одному» имеет уникальный индекс по внешнему ключу `BookId`, и `PriceOffer` (мягко удаленная) занимает этот слот.

Как выяснили мои клиенты, функция мягкого удаления полезна, потому что пользователи могут по ошибке удалить не те данные. Но осведомленность о проблемах позволяет планировать, как их решать в своих приложениях. Обычно я использую подход «Корень/Агрегат» и не допускаю мягкое удаление зависимых сущностей со связью «один к одному».

ИСПОЛЬЗОВАНИЕ ФИЛЬТРОВ ЗАПРОСОВ ДЛЯ СОЗДАНИЯ МУЛЬТИТЕНАНТНЫХ СИСТЕМ

Мультитенантная система (*multitenant system*) – это система, в которой разные пользователи или группы пользователей имеют данные, доступ к которым предоставлен только определенным пользователям. Можно найти множество примеров, например Office365 и GitHub. Одной функциональности фильтров запросов недостаточно для создания Office365, но можно использовать эти фильтры для создания сложного мультитенантного приложения.

При использовании фильтра запросов с мягким удалением мы использовали логическое значение в качестве фильтра, но для мультитенантной системы нужен более сложный ключ, который я называю DataKey. У каждого клиента (tenant) есть уникальный ключ DataKey. Клиентом может быть отдельный пользователь или, что более вероятно, группа пользователей. На рис. 6.5 показан пример SaaS-приложения, обеспечивающего контроль запасов для большого числа розничных компаний. В данном случае Джо работает на Dress4U и у него есть ключ DataKey для входа в систему.

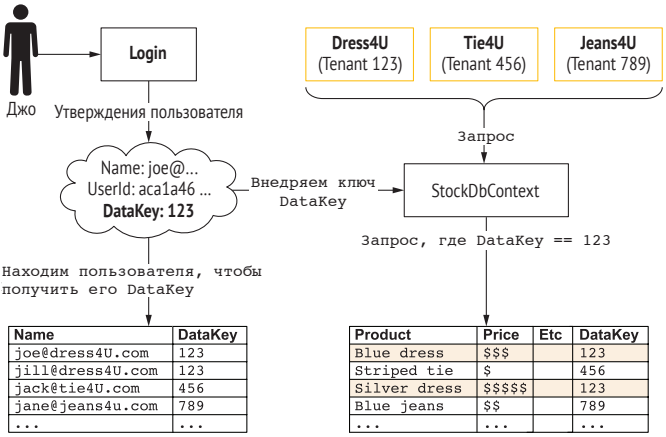


Рис. 6.5 Когда Джо выполняет вход, его имя и `UserId` ищутся в таблице `DataKeyLookup`, а соответствующий ключ DataKey (123) добавляется к его утверждениям (user claims). Когда Джо запрашивает список товаров, ключ DataKey из утверждений извлекается и передается в DbContext при его создании. Затем этот ключ используется в глобальном фильтре запросов, примененном к таблице Stock. Таким образом, Джо видит только Blue dress и Silver dress

В приложении Book App никому не нужно выполнять вход, поэтому здесь нельзя реализовать подход, показанный на рис. 6.5, но у него есть cookie-корзина с псевдоидентификатором `UserId`, который мы можем использовать. Когда пользователь выбирает книгу для покупки в приложении, создается cookie-корзина, чтобы хранить книги в корзине пользователя, а также `UserId`.

Cookie-корзина используется, если пользователь выбирает пункт меню «Мои заказы», чтобы увидеть только заказы этого пользователя. Следующий код берет `UserId` из cookie-корзины и использует фильтр запросов для возврата только тех заказов, которые были созданы пользователем. Этот код работает благодаря двум основным частям:

- `UserIdService` получает `UserId` из cookie-корзины;
- `IUserIdService` внедряется через конструктор DbContext и используется для доступа к текущему пользователю.

В следующем листинге показан код `UserIdService`, использующий `IHttpContextAccessor` для доступа к текущему HTTP-запросу.

Листинг 6.4 Класс `UserIdService`, извлекающий `UserId` из cookie-корзины

```
public class UserIdService : IUserIdService
{
    private readonly IHttpContextAccessor _httpAccessor;

    public UserIdService(IHttpContextAccessor httpAccessor)
    {
        _httpAccessor = httpAccessor;
    }

    public Guid GetUserId()
    {
        var httpContext = _httpAccessor.HttpContext;
        if (httpContext == null)
            return Guid.Empty;

        var cookie = new BasketCookie(httpContext.Request.Cookies);
        if (!cookie.Exists())
            return Guid.Empty;

        var service = new CheckoutCookieService(cookie.GetValue());
        return service.UserId;
    }
}
```

IHttpContextAccessor – это способ доступа к текущему HTTPContext. Чтобы использовать его, необходимо зарегистрировать его в классе Startup, применяя команду services.AddHttpContextAccessor()

В некоторых случаях HTTPContext может иметь значение null, например в фоновой задаче. В таком случае предоставляется пустой GUID

Если cookie-корзина есть, создает CheckoutCookieService, который извлекает UserId и возвращает его

Использует существующие сервисы для поиска cookie-корзины. Если cookie нет, код возвращает пустой идентификатор GUID

Если у вас есть значение, которое будет действовать как ключ DataKey, то необходимо передать его в DbContext. Распространенный способ – внедрение через конструктор; внедренный сервис предоставляет способ получить этот ключ. В нашем примере мы используем `UserId`, взятый из cookie-корзины в качестве этого ключа. Затем используем этот `UserId` в фильтре запроса, примененном к свойству `CustomerId` в классе сущности `Order`. Он содержит `UserId` человека, создавшего заказ. Любой запрос сущностей `Order` вернет только заказы, созданные текущим пользователем. В следующем листинге показано, как внедрить сервис `UserIdService` в DbContext, а затем использовать этот `UserId` в фильтре запросов.

Листинг 6.5 DbContext в приложении Book App с внедрением `UserId` и фильтром запроса

```
public class EfCoreContext : DbContext
{
    private readonly Guid _userId;

    public EfCoreContext(DbContextOptions<EfCoreContext> options,
```

Это свойство содержит UserId, используемый в фильтре запросов в классе сущности Order

Обычные параметры для настройки DbContext

```
Настраивает UserId. Если UserId имеет значение NULL, версия с простой заменой предоставляет значение Guid.Empty по умолчанию

Настраивает UserIdService. Обратите внимание: этот параметр является необязательным, что значительно упрощает его использование в модульных тестах, в которых не используется фильтр запросов

            IUserIdService userIdService = null)
            : base(options)
        {
            _userId = userIdService?.GetUserId()
                      ?? new ReplacementUserIdService().GetUserId();
        }

Метод, в котором вы настраиваете EF Core и устанавливаете фильтры запросов

        public DbSet<Book> Books { get; set; }
        //… остальной код DbSet<T> опущен

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            //… Остальная конфигурация опущена для ясности;

            modelBuilder.Entity<Book>()
                .HasQueryFilter(p => !p.SoftDeleted);

Фильтр запросов с мягким удалением

            modelBuilder.Entity<Order>()
                .HasQueryFilter(x => x.CustomerName == _userId);

Фильтр запроса заказа, который соответствует текущему UserId, полученному из cookie-корзины, с CustomerId в классе сущности Order

        }
    }
```

Для ясности: каждый экземпляр DbContext получает `UserId` текущего пользователя или пустой GUID, если он никогда не «покупал» книгу. В то время как конфигурация DbContext настраивается при первом использовании и кешируется, лямбда-выражение фильтра запроса привязано к активному полю _userId. Фильтр запросов фиксированный, но _userId является динамическим и может меняться в каждом экземпляре DbContext.

Однако важно, чтобы фильтр запросов не был помещен в отдельный конфигурационный класс (см. раздел 7.5.1), потому что _userId станет фиксированным для `UserId`, предоставленного при первом использовании. Нужно поместить лямбда-запрос куда-нибудь, чтобы он мог получить динамическую переменную _userId. В этом случае я помещаю его в метод `OnModelCreating` внутри DbContext-приложения, и это нормально. В главе 7 я покажу вам способ автоматизации настройки фильтров запросов, которые поддерживают динамический _userId; см. раздел 7.15.4.

Если у вас есть приложение ASP.NET Core, в котором авторизуются пользователи, можно использовать `IHttpContextAccessor` для доступа к текущему `ClaimPrincipal`. `ClaimPrincipal` содержит список утверждений для вошедшего в систему пользователя, включая его `UserId`, который хранится в утверждении с именем, определенным системной константой `ClaimTypes.NameIdentifier`. Или, как показано на рис. 6.5, можно добавить новое утверждение пользователю при входе в систему, чтобы предоставить ключ DataKey, используемый в фильтре запросов.

ПРИМЕЧАНИЕ Пример полноценной мультитенантной системы, в которой используется идентификатор пользователя, чтобы найти ключ клиента DataKey во время входа в систему, а утверждение DataKey добавляется к утверждениям пользователя, можно найти в статье на странице http://mng.bz/yY7q.

6.1.7 Команды LINQ, требующие особого внимания

EF Core отлично справляется с отображением методов LINQ в SQL, языке большинства реляционных баз данных. Но есть три типа методов LINQ, требующих особого внимания:

- команды, для которых необходим дополнительный код, чтобы соответствовать принципам работы базы данных, например `Average`, `Sum`, `Max` и другие агрегатные команды, требующие обработки возврата `null`. Практически единственный агрегат, который не возвращает значение `null`, – это `Count`;
- команды, которые могут работать с базой данных, но только с жесткими ограничениями, потому что база данных не поддерживает все возможности команды. Пример – команда `GroupBy`: у базы данных может быть только простой ключ, а для `IGrouping` есть существенные ограничения;
- команды, соответствующие функциям базы данных, но с некоторыми ограничениями на то, что может возвращать база данных, например команды `Join` и `GroupJoin`.

В документации по EF Core есть отличная статья под названием «Операторы сложных запросов» (http://mng.bz/MXan) с прекрасными описаниями многих из этих команд, поэтому я не буду рассматривать их все. Но хочу предупредить вас об опасном исключении `InvalidOperationException` с сообщением, содержащим слова `could not be translated`, и рассказать, что делать, если вы его получите.

Проблема состоит в том, что если вы ошибетесь в LINQ, то получите сообщение `could not be translated`. Возможно, оно не слишком полезно при диагностике проблемы (но см. следующее примечание), кроме сообщения о том, что вам следует: `switch to client evaluation explicitly by inserting a call to AsEnumerable ...`". Хотя и можно переключиться на вычисление на стороне клиента, вы можете получить (большое) снижение производительности.

ПРИМЕЧАНИЕ Команда EF Core уточняет сообщения, возвращаемые из исключения `could not be translated`, и добавляет конкретные сообщения для распространенных ситуаций, например попытка использовать метод `String.Equal` с параметром `StringComparison` (который не может быть преобразован в SQL).

В следующем разделе представлено несколько советов, как заставить наиболее распространенные сложные команды работать с реля-

ционной базой данных. Также предлагаю вам протестировать сложные запросы, так как в них легко ошибиться.

АГРЕГАТАМ НУЖЕН NULL (КРОМЕ COUNT)

Вы, вероятно, будете использовать агрегаты LINQ Max, `Min`, Sum, `Average`, `Count` и `CountLong`. Вот несколько советов, как заставить их работать:

- методы `Count` и `CountLong` прекрасно работают, если вы ведете подсчет чего-то разумного в базе данных, например если это строка или относительные ссылки, допустим количество отзывов на книгу;
- для агрегатов `Max`, `Min`, `Sum` и `Average` требуется результат, допускающий значение NULL, например `context.Books.Max(x => (decimal?)x.Price)`. Если источник (в этом примере это `Price`) не допускает значение NULL, необходимо приведение к типу, допускающему это значение. Кроме того, если вы используете библиотеку Sqlite для модульного тестирования, помните, что она не поддерживает `decimal`, поэтому вы получите сообщение об ошибке, даже если использовали версию, допускающую значение NULL;
- нельзя использовать метод `Aggregate` непосредственно в базе данных, потому что он выполняет расчет для каждой строки.

КОМАНДА GROUPBY

Еще один метод LINQ, который может оказаться полезен, – это `GroupBy`. Когда он используется с базой данных SQL, часть `Key` должна быть скалярным значением (или значениями), потому что это то, что поддерживает оператор SQL `GROUP BY`. Часть `IGrouping` может быть набором данных, в том числе командами LINQ. Мой опыт показывает, что нужно использовать команду `GroupBy` вместе с командой материализации (см. раздел 2.3.3), такой как `ToList`. Все остальное вызывает исключение `could not be translated`.

Вот реальный пример, взятый из клиентского приложения. Некоторые имена изменены, чтобы сохранить конфиденциальные данные клиента. Обратите внимание, что `Key` может быть комбинацией скалярных столбцов и части `IGrouping`:

```
var something = await _context.SomeComplexEntity
    .GroupBy(x => new { x.ItemID, x.Item.Name })
    .Select(x => new
    {
          Id = x.Key.ItemID,
          Name = x.Key.Name,
          MaxPrice = x.Max(o => (decimal?)o.Price)
    })
   .ToListAsync();
```

6.1.8 Использование AutoMapper для автоматического построения запросов с методом Select

В главе 2 мы узнали, что выборочные запросы (см. 2.4.3) позволяют создать один запрос, возвращающий именно те данные, которые нам необходимы. Эти запросы часто весьма эффективны с точки зрения производительности. Проблема состоит в том, что они отнимают больше времени при написании – всего на несколько строк больше, но в случае с реальными приложениями запросов могут быть тысячи, поэтому каждый выборочный запрос увеличивает время разработки. Я всегда ищу способы автоматизации, и AutoMapper (https://automapper.org) может помочь автоматизировать построение такого рода запросов.

Я не буду описывать все особенности данной библиотеки. Для этого может потребоваться целая книга! Но я приведу обзор, где рассказывается, как настроить и использовать AutoMapper, поскольку я не думаю, чтобы эти темы были достаточно освещены где-то еще. Начнем со сравнения простого выборочного запроса, код которого пишется вручную, с таким же запросом, написанным с помощью AutoMapper, как показано на рис. 6.6.

Версия с кодом, написанным вручную

```
var dto = context.Books
    .Select(p => new ChangePubDateDto
    {
        BookId = p.BookId,
        Title = p.Title,
        PublishedOn = p.PublishedOn
    })
    .Single(k => k.BookId == lastBook.BookId);
```

Версия с AutoMapper

```
var dto = context.Books
    .ProjectTo<ChangePubDateDtoAm>(config)
    .Single(x => x.BookId == lastBook.BookId);
```

Рис. 6.6 Обе версии выборочного запроса дают одинаковые результаты и одинаковый код SQL. Это очень простой запрос, и скопированы только три свойства, но он дает представление о том, как работает AutoMapper. В данном случае у DTO свойства того же типа и с теми же именами, что и свойства, которые мы хотим скопировать. Это означает, что AutoMapper автоматически создаст код LINQ для копирования этих трех свойств

Несмотря на простоту примера, он показывает, что можно уменьшить код такого запроса до одной строки с помощью метода AutoMapper, `ProjectTo`. Здесь используется конфигурация «По соглашению», где свойства из источника – в данном случае класса `Book` – отображаются в свойства DTO путем сопоставления их по типу и имени каждого свойства. AutoMapper может автоматически отображать некоторые связи. Например, свойство типа `decimal` с именем `PromotionNewPrice` будет отображать связь `Promotion.NewPrice`. (В AutoMapper это называется *сглаживанием* – http://mng.bz/aorB.)

На рис. 6.7 показаны четыре способа конфигурации «По соглашению» с использованием AutoMapper:

- *отображение по типу и имени* – свойства отображаются из класса сущности в одноименные свойства DTO с тем же типом;
- *отсечение свойств* – если вы убираете свойства, которые находятся в классе сущности из DTO, то выборочный запрос не будет загружать эти столбцы;
- *сглаживание отношений* – имя в DTO представляет собой комбинацию имени навигационного свойства и свойства в типе навигационного свойства. `Promotion.NewPrice` из сущности `Book`, например, отображается в свойство DTO `PromotionNewPrice`;
- *вложенные DTO* – эта конфигурация позволяет отображать коллекции из класса сущности в класс DTO, поэтому вы можете скопировать определенные свойства из класса сущности в навигационное свойство коллекции.

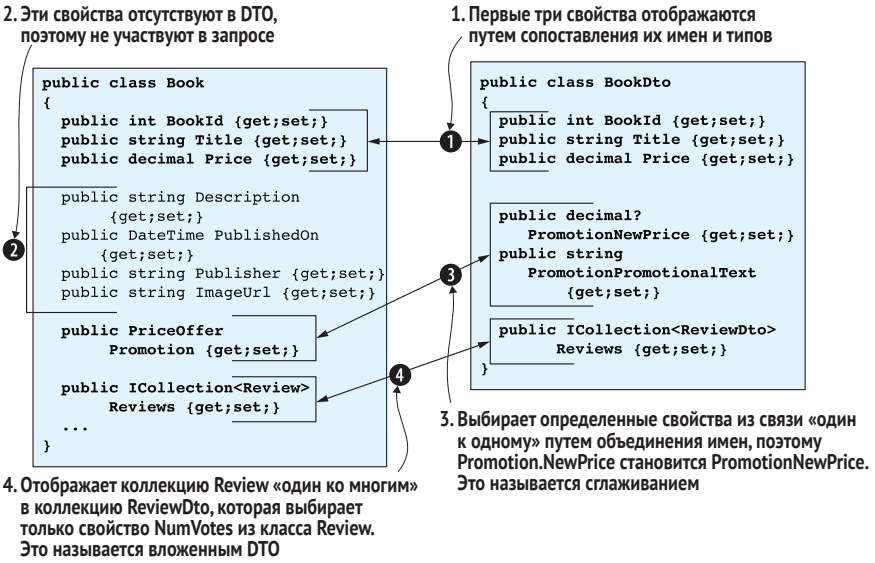


Рис. 6.7 Четыре способа, с помощью которых AutoMapper автоматически отображает класс сущности `Book` в класс `BookDto`. Соглашение по умолчанию заключается в том, чтобы выполнять отображение, используя одинаковые имена и типы, включая обработку связей, используя эквивалентное пути к свойству имя, но без точек. Свойство DTO `PromotionNewPrice`, например, автоматически отображается в свойство `Promotion.NewPrice`. Отображения также могут быть вложенными – коллекцию из класса сущности можно отобразить в коллекцию из DTO

Теперь, когда у вас есть представление о том, что может делать AutoMapper, хочу дать вам несколько советов по поводу того, как настроить и использовать его.

ДЛЯ ПРОСТОГО ОТОБРАЖЕНИЯ ИСПОЛЬЗУЙТЕ АТРИБУТ [AUTOMAP]

Использовать метод AutoMapper `ProjectTo` просто, но он опирается на конфигурацию AutoMapper, которая сложнее. В версии 8.1 Джимми

Богарт добавил атрибут `AutoMap`, который допускает конфигурацию «По соглашению» простых отображений. В следующем фрагменте кода этот атрибут показан в первой строке (выделена жирным шрифтом), где мы определяем, из какого класса сущности этот DTO должен отображаться:

```
[AutoMap(typeof(Book))]
public class ChangePubDateDtoAm
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
}
```

Классы, отображаемые с помощью атрибута `AutoMap`, используют конфигурацию AutoMapper «По соглашению», с несколькими параметрами и атрибутами для дополнительной настройки. Как было показано на рис. 6.7, такая конфигурация может сделать многое, но, безусловно, не все, что вам может понадобиться. Для этого нужен класс AutoMapper `Profile`.

Для сложных отображений требуется класс Profile

Когда подхода с использованием конфигурации «По соглашению» недостаточно, нужно создать класс AutoMapper `Profile`, позволяющий определять отображение для свойств, не охваченных этим подходом. Чтобы отобразить `Book` в `BookListDto` из листингов 2.10 и 2.11, например, трем из девяти свойств DTO нужна специальная обработка. Нужно создать `MappingConfiguration`. Для этого есть несколько способов, но обычно используется класс `Profile`, который легко найти и зарегистрировать. В следующем листинге показан класс, наследующий от класса `Profile` и устанавливающий отображения, которые AutoMapper слишком сложно сформировать.

Листинг 6.6 Класс `Profile` с конфигурацией специальных отображений для некоторых свойств

```
public class BookListDtoProfile : Profile
{
    public BookListDtoProfile()
    {
        CreateMap<Book, BookListDto>()
            .ForMember(p => p.ActualPrice,
                m => m.MapFrom(s => s.Promotion == null
                ? s.Price : s.Promotion.NewPrice))
            .ForMember(p => p.AuthorsOrdered,
                m => m.MapFrom(s => string.Join(", ",
                        s.AuthorsLink.Select(x => x.Author.Name))));
    }
}
```

Ваш класс должен наследовать от класса Profile. У вас может быть несколько классов, которые наследуют от него

Устанавливает отображение из класса сущности Book в BookListDto

Фактическая цена (ActualPrice) зависит от того, есть ли Promotion с NewPrice

Получает список имен авторов в качестве строки, разделенной запятыми

```
            .ForMember(p => p.ReviewsAverageVotes,
                m => m.MapFrom(s =>
                    s.Reviews.Select(y =>
                        (double?)y.NumStars).Average()));
    }
}
```

Содержит специальный код, необходимый для создания метода Average, выполняемого в базе данных

Здесь мы настраиваем три из девяти свойств, а для остальных шести свойств используется подход «По соглашению», поэтому некоторые из имен свойств в классе `ListBookDto` длинные. Например, у свойства DTO `PromotionPromotionalText` такое имя, потому что по соглашению оно отображается в навигационное свойство `Promotion`, а затем в свойство `PromotionalText` в классе сущности `PriceOffer`.

Вы можете добавить множество вызовов `CreateMap` в одном профиле, или у вас может быть несколько профилей. Профили могут быть сложными, и управление ими – главное больное место при использовании AutoMapper. У одного из моих клиентов был класс `Profile` длиной в 1000 строк.

РЕГИСТРАЦИЯ КОНФИГУРАЦИЙ AUTOMAPPER

На последнем этапе все отображения регистрируются с использованием внедрения зависимостей. К счастью, у AutoMapper есть пакет NuGet `AutoMapper.Extensions.Microsoft.DependencyInjection`, содержащий метод `AddAutoMapper`. Он сканирует предоставляемые сборки и регистрирует интерфейс `IMapper` в качестве сервиса. Этот интерфейс используется для внедрения конфигурации для всех ваших классов, у которых есть атрибут `[AutoMap]`, и всех классов, которые наследуют от класса `Profile`. В приложении ASP.NET Core следующий фрагмент кода будет добавлен в метод `Configure` класса `Startup`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    // … Остальной код удален для ясности;

    services.AddAutoMapper( MyAssemblyToScan1, MyAssemblyToScan2…);
}
```

6.1.9 Оценка того, как EF Core создает класс сущности при чтении данных

До сих пор классы сущностей в этой книге не имели пользовательских конструкторов. При чтении такого класса EF Core использует конструктор без параметров по умолчанию, а затем обновляет свойства и резервные поля напрямую. (Резервные поля описаны в главе 7.) Но иногда полезно иметь конструктор с параметрами, потому что это облегчает создание экземпляра, например когда вы хотите гарантировать, что класс создан правильно.

ПРИМЕЧАНИЕ Использование конструкторов для создания класса – хороший подход, потому что вы можете определить, каким параметрам нужно установить значение, чтобы создать допустимый экземпляр. При использовании DDD с EF Core (см. главу 13) единственный способ создать класс сущности – использовать конструктор или статическую фабрику.

С момента появления версии 2.1 в EF Core используется конструктор класса сущности, если необходимо создать экземпляр такого класса (как правило, при чтении данных). Если вы используете для своего конструктора паттерн «По соглашению», т. е. параметры конструктора совпадают со свойствами по типу и имени (camelCase/PascalCase) и не включают навигационные свойства, как показано в следующем листинге, EF Core также будет его использовать.

Листинг 6.7 Класс сущности с конструктором, который работает с EF Core

```
public class ReviewGood
{
    public int Id { get; private set; }
    public string VoterName { get; private set; }
    public int NumStars { get; set; }

    public ReviewGood
        (string voterName)
    {
        VoterName = voterName;
        NumStars = 2;
    }
}
```

Можно установить для свойств закрытый метод доступа set. EF Core по-прежнему сможет их записывать

Конструктору не нужны параметры для всех свойств в классе. Кроме того, тип доступности конструктора может быть любым: public, private и т. д.

EF Core будет искать параметр с тем же типом и именем, которое соответствует свойству (используя camelCase или PascalCase)

Присваивание не должно включать в себя какое-либо изменение данных; в противном случае вы не получите точные данные, которые были в базе

Любое присваивание свойству, для которого нет параметра, – это нормально. EF Core установит это свойство после исполнения конструктора в значение, прочитанное из базы

Можно было бы добавить конструктор в класс `ReviewGood`, который устанавливает для всех свойств значения, не являющиеся навигационными, но я хотел показать, что EF Core может использовать конструктор для создания экземпляра сущности, а затем установить значения всех свойств, которые не были переданы в параметрах конструктора. Теперь, посмотрев на рабочий конструктор, рассмотрим конструкторы, которые EF Core не может или не будет использовать, и как справиться с этими проблемами.

КОНСТРУКТОРЫ, КОТОРЫЕ МОГУТ ВЫЗВАТЬ У ВАС ПРОБЛЕМЫ С EF CORE

Первый тип конструктора, который не может использовать EF Core, – конструктор с параметром, чей тип или имя не совпадает со свой-

ствами класса. В следующем листинге показан пример с параметром `starRating`, который назначается свойству `NumStars`. Если это единственный конструктор, то EF Core выбросит исключение при первом использовании DbContext.

Листинг 6.8 Класс с конструктором, который EF Core не может использовать и вызовет исключение

```
public class ReviewBadCtor
{
    public int Id { get; set; }
    public string VoterName { get; set; }
    public int NumStars { get; set; }

    public ReviewBadCtor(                    <-- Единственный конструктор в этом классе
        string voterName,
        int starRating)                      <-- Имя этого параметра не соответствует названию
    {                                            ни одного из свойств в этом классе, поэтому
        VoterName = voterName;                   EF Core не может использовать его для создания
        NumStars = starRating;                   экземпляра класса при чтении данных
    }
}
```

Еще один пример конструктора, который не может использовать EF Core, – это конструктор с параметром, задающим навигационное свойство. Например, если бы у класса сущности `Book` был конструктор, который включал параметр для установки навигационного свойства `PriceOffer Promotion`, то EF Core не смог бы его использовать. Конструктор, который может использовать EF Core, должен иметь только нереляционные свойства.

Если ваш конструктор не соответствует паттерну «По соглашению», то необходимо предоставить конструктор, который EF Core может использовать. Стандартное решение – добавить закрытый конструктор без параметров, который EF Core может использовать для создания экземпляра класса, а затем использовать свой обычный способ записи параметров/полей.

> **ПРИМЕЧАНИЕ** EF Core может использовать конструкторы с модификаторами доступа. Он использует любой уровень доступа, от закрытых (`private`) до открытых (`public`) конструкторов. Как уже было показано, он также может записать значение в свойство, защищенное от записи, например `public int Id {get; private set;}`. EF Core может обрабатывать свойства с доступом только на чтение (например, экземпляр `public int Id {get;}`), но с некоторыми ограничениями (см. http://mng.bz/go2e).

Еще одна, более тонкая проблема возникает, если изменять значения при присваивании их соответствующему свойству. Следующий фрагмент кода вызвал бы проблемы, потому что полученное значение будет изменено при присваивании:

```
public ReviewBad(string voterName)
{
    VoterName = "Name: " + voterName; // Изменяем параметр, прежде чем назначить
      его свойству;
    //… Остальной код пропущен;
}
```

Результат присваивания в конструкторе ReviewBad означает, что если данные в базе данных – это XXX, то после чтения было бы Name: XXX, а это не то, что нам нужно. Решение состоит в том, чтобы изменить имя параметра, чтобы он не соответствовал имени свойства. В данном случае можно назвать его `voterNameNeedingPrefix`.

Наконец, имейте в виду, что все проверки и валидации, которые вы применяете к параметрам конструктора, будут применяться, когда EF Core использует конструктор. Если у вас есть проверка, что строка не равна NULL, то следует настроить столбец базы данных так, чтобы он не мог содержать NULL (см. главу 7) или чтобы неконтролируемые данные в базе данных не возвращали значение NULL.

EF CORE МОЖЕТ ВНЕДРЯТЬ ОПРЕДЕЛЕННЫЕ СЕРВИСЫ, ИСПОЛЬЗУЯ КОНСТРУКТОР СУЩНОСТИ

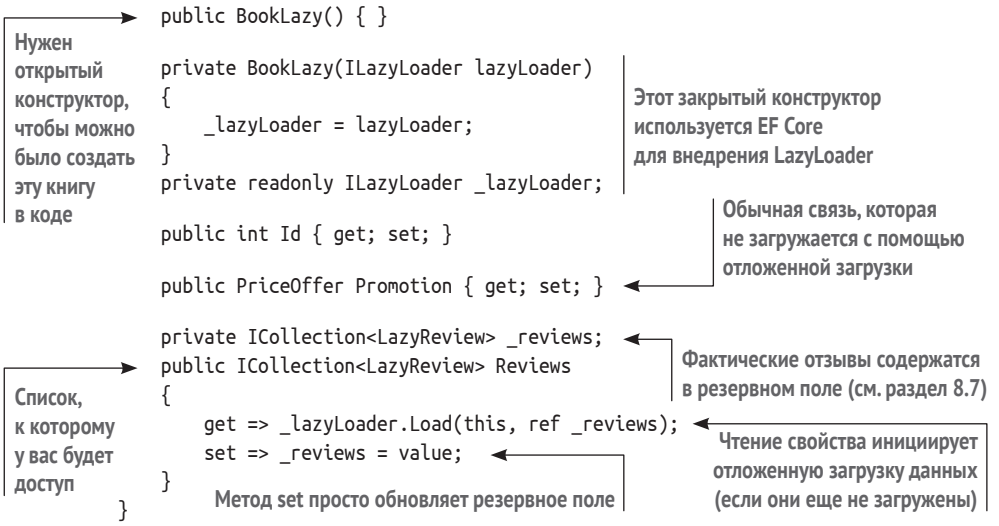
В то время как мы говорим о конструкторах, мы должны обратить внимание на способность EF Core внедрять сервисы через конструктор класса сущности. EF Core может внедрять три типа сервисов, наиболее полезный из которых внедряет метод для отложенной загрузки связей, который я опишу в полном объеме. Два других варианта являются расширенными функциями; я кратко опишу, что они делают, и предоставлю ссылку на документацию по EF Core для получения дополнительной информации.

Из раздела 2.4.4 вы узнали, как настроить отложенную загрузку связей через пакет NuGet `Microsoft.EntityFrameworkCore.Proxies`. Это самый простой способ настроить отложенную загрузку, но у него есть недостаток: все навигационные свойства должны быть настроены на использование отложенной загрузки, т. е. у каждого навигационного свойства должно быть указано ключевое слово `virtual`.

Если вы хотите ограничить то, какие связи используют отложенную загрузку, то можно получить сервис отложенной загрузки с помощью конструктора класса сущности. Затем вы изменяете навигационные свойства так, чтобы использовать этот сервис в методе чтения свойства. В следующем листинге показан класс сущности `BookLazy`, у которого две связи: `PriceOffer`, которая не использует отложенную загрузку, и `Reviews`, которая использует ее.

Листинг 6.9 Как работает отложенная загрузка через внедренный метод lazyLoader

```
public class BookLazy
{
```



Для внедрения сервиса через интерфейс `Ilazyloader` нужно добавить в проект пакет NuGet `Microsoft.EntityFrameworkcore.Abstractions`. У него имеется минимальный набор типов и нет зависимостей, поэтому он не будет «загрязнять» проект ссылками на DBContext и другие типы доступа к данным.

Но если вы придерживаетесь архитектуры, которая не допускает присутствия никаких внешних пакетов, то можно добавить в конструктор сущности параметр типа `Action<object, string>`. EF Core заполнит этот параметр, используя действие, которое принимает экземпляр сущности в качестве своего первого параметра и имя поля как второй параметр. Когда это действие вызывается, оно загружает данные связей в именованное поле в указанном экземпляре класса сущности.

> **ПРИМЕЧАНИЕ** С помощью дополнительного небольшого метода расширения можно заставить `Action<object, string>` работать аналогично `Ilazyloader`. Этот эффект можно увидеть в методе расширения в конце раздела Lazy loading without proxies на странице документации EF Core по адресу http://mng.bz/e5zv в классе `LazyBook2` в проекте `Test` в репозитории GitHub для этой книги.

Есть еще два способа внедрения сервиса в класс сущности через конструктор:

- внедрение экземпляра DbContext, с которым связан класс сущности, полезно, если вы хотите выполнять доступ к базе данных внутри своего класса сущности. Я расскажу об этом в главе 13. Если вкратце, то вам не следует использовать эту технику, если у вас нет серьезной проблемы с производительностью или бизнес-логикой, которую нельзя решить никаким другим способом;

- `IEntityType` для этого экземпляра класса сущности дает доступ к конфигурации, `State`, информации об этой сущности и другим данным, связанным с этим типом сущности.

Эти два метода являются расширенными возможностями, и я не буду подробно их рассматривать. В документации EF Core по конструкторам классов сущностей содержится дополнительная информация по данной теме; см. http://mng.bz/pV78.

6.2 Запись данных в базу с EF Core

Первая часть этой главы была посвящена запросам к базе данных. Теперь мы перейдем к записи данных в базу: созданию, обновлению и удалению классов сущностей. Как и в разделе 6.1, моя цель – показать вам внутреннюю работу EF Core при выполнении записи данных в базу. Некоторые подразделы раздела 6.2 посвящены изучению того, что происходит, когда вы записываете данные в базу, а некоторые – изящным методам быстрого копирования или удаления данных. Вот список тем, о которых пойдет речь:

- оценка того, как EF Core записывает сущности со связями в базу данных;
- оценка того, как DbContext обрабатывает запись сущностей со связями;
- быстрое копирование данных со связями;
- быстрое удаление сущности.

6.2.1 Оценка того, как EF Core записывает сущности или связи в базу данных

Когда вы создаете новую сущность с новой связью (связями), навигационные свойства – ваши друзья, потому что EF Core берет на себя задачу заполнения внешнего ключа. В следующем листинге показан простой пример: добавление новой книги с новым отзывом.

Листинг 6.10 Добавление новой сущности Book с новым отзывом

```
var book = new Book
{
    Title = "Test",
    Reviews = new List<Review>()
};
book.Reviews.Add(
    new Review { NumStars = 1 });
context.Add(book);
context.SaveChanges();
```

Создает новую книгу

Добавляет новый отзыв в навигационное свойство Reviews

Метод Add говорит, что экземпляр сущности нужно добавить в соответствующую строку со всеми добавленными или обновленными связями

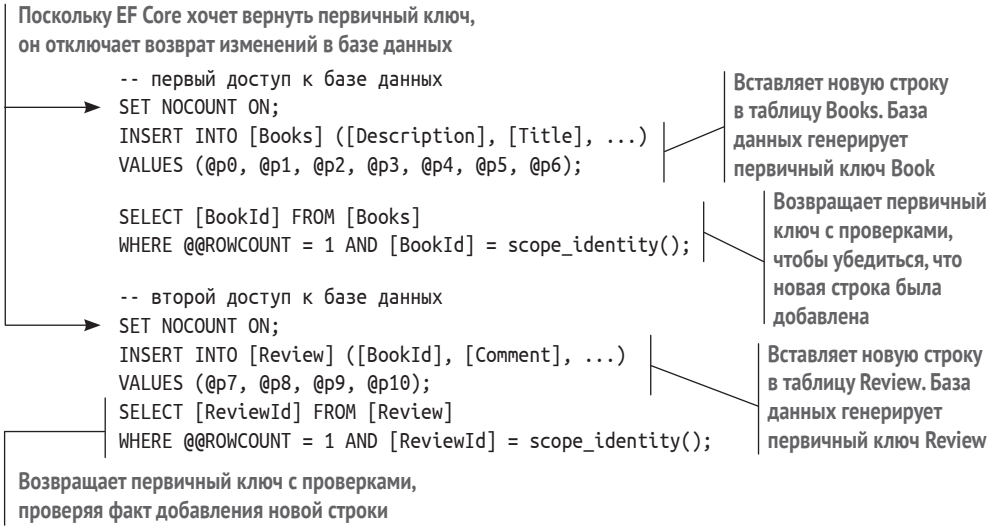
Метод SaveChanges выполняет обновление базы данных

Чтобы добавить эти две связанные сущности в базу данных, EF Core должен сделать следующее:

- *определить порядок, в котором нужно создавать новые строки,* – в данном случае нужно создать строку в таблице Books, чтобы у нее был первичный ключ `Book`;
- *скопировать все первичные ключи во внешние ключи связей* – в данном случае копируется первичный ключ строки таблицы Books, `BookId`, во внешний ключ в новой строке Review;
- *скопировать все новые данные, созданные в базе данных, чтобы классы сущностей правильно представляли базу данных,* – здесь необходимо скопировать обратно `BookId` и обновить свойство `BookId` в классах сущностей `Book` и `Review`, а также `ReviewId` для класса сущности `Review`.

В следующем листинге показаны команды SQL для создания двух строк.

Листинг 6.11 Команды SQL для создания двух строк с возвратом первичных ключей



Это простой пример, но он охватывает все основные моменты. Нужно понять вот что: вы можете создавать сложные данные со связями и связями этих связей, а EF Core решит, как добавить их в базу данных.

Я встречал код, в котором разработчик использовал несколько вызовов метода `SaveChanges` для получения первичного ключа из первой операции создания, чтобы установить внешний ключ для связанной сущности. В этом нет необходимости, если у вас есть навигационные свойства, связывающие разные сущности. Поэтому если вы считаете, что нужно вызвать метод `SaveChanges` дважды, значит, вы не настроили правильные навигационные свойства для обработки этого случая.

ПРЕДУПРЕЖДЕНИЕ Не рекомендуется вызывать метод `SaveChanges` несколько раз для создания сущностей со связями, потому что если по какой-то причине второй метод `SaveChanges` не сработает, у вас будет неполный набор данных в базе данных, что может привести к проблемам. Смотрите врезку «Почему нужно вызывать метод `SaveChanges` только один раз в конце ваших изменений» из раздела 3.2.2 для получения дополнительной информации.

6.2.2 Оценка того, как DbContext обрабатывает запись сущностей и связей

В разделе 6.2.1 вы видели, что EF Core делает на стороне базы данных, а теперь мы посмотрим, что происходит внутри EF Core. В большинстве случаев эта информация вам не нужна, но иногда важно знать ее. Если во время вызова метода `SaveChanges` вы перехватываете изменения, то, например, вы получите его State только перед вызовом этого метода, а первичный ключ вновь созданной сущности появится у вас только после вызова.

ПРИМЕЧАНИЕ Я столкнулся с этой проблемой, когда готовил первое издание этой книги. Мне нужно было обнаружить изменения в классе сущности `Book` и изменения всех связанных с ней классов сущностей, таких как `Review`, `BookAuthor` и `PriceOffer`. В тот момент мне нужно было перехватить State каждой сущности в начале, но у меня могло не быть правильного внешнего ключа до того, как метод `SaveChanges` завершал работу.

Даже если вы не пытаетесь решить сложную задачу, подобную этой, полезно понять, как работает EF Core. Следующий пример немного сложнее предыдущего, потому что я хочу показать вам разные способы, используя которые, EF Core обрабатывает новые экземпляры класса сущности поверх экземпляра сущности, который был прочитан из базы данных. В следующем листинге мы создаем новую сущность `Book`, но с `Author`, который уже есть в базе данных. Код поделен на этапы: `ЭТАП 1`, `ЭТАП 2` и `ЭТАП 3` – и дополнен пояснениями после каждого этапа.

Листинг 6.12 Создание новой сущности Book с новой связью «многие ко многим» с существующим автором

```
//ЭТАП 1
var author = context.Authors.First();
var bookAuthor = new BookAuthor { Author = author };
```

Каждый из трех этапов начинается с комментария

Создает новую строку BookAuthor и готов привязать Book к Author

Читает существующую сущность Author для новой книги

```
var book = new Book
{
    Title = "Test Book",
    AuthorsLink = new List<BookAuthor> { bookAuthor }
};
```

Создает книгу и заполняет навигационное свойство AuthorsLink одной записью, связывая ее с существующим автором

```
//ЭТАП 2
context.Add(book);
```

Вызывает метод Add, сообщающий EF Core, что книгу необходимо добавить в базу данных

```
//ЭТАП 3
context.SaveChanges();
```

Метод SaveChanges просматривает все отслеживаемые сущности и выясняет, как обновить базу данных, чтобы добиться желаемого

На рис. 6.8–6.10 показано, что происходит внутри классов сущностей и отслеживаемые данные на каждом этапе. На каждом из трех рисунков показаны следующие данные в конце этапа:

- свойство `State` каждого экземпляра сущности на каждом этапе (показано над каждым классом сущности);
- первичный и внешний ключи с текущим значением в скобках. Если значение ключа – (0), то он еще не задан;
- навигационные ссылки отображаются как соединения из навигационного свойства к соответствующему классу сущности, с которым он связан;
- изменения между этапами отображаются жирным шрифтом или более толстыми линиями для навигационных связей.

На рис. 6.8 показана ситуация после завершения этапа 1. Этот исходный код устанавливает связь нового класса сущности `Book` (слева) с новым классом сущности `BookAuthor` (посередине), который связывает Book с существующим классом сущности `Author` (справа).

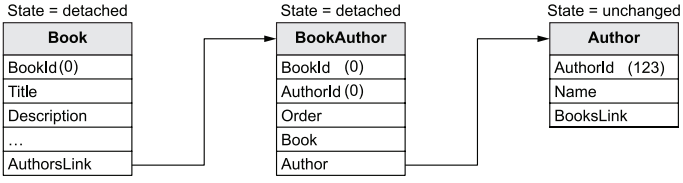


Рис. 6.8 Конец этапа 1. На этом рисунке показано, что новый класс сущности `Book` с новым классом сущности `BookAuthor`, связанным с `Book`, имеют `State=Detached`, а существующий класс сущности `Author`, который был считан из базы данных, имеет `State=Unchanged`. На рисунке также показаны две навигационные связи, которые были установлены для связи сущности `Book` с сущностью `Author`. Наконец, первичный и внешний ключи `Book` и `BookAuthor` не заданы, т. е. равны нулю, тогда как у сущности `Author` имеется первичный ключ (123), потому что она уже находится в базе данных

На рис. 6.8 представлена графическая версия трех классов сущностей, после того как завершился этап 1 из листинга 6.12. Этот рисунок – отправная точка перед вызовом любых методов EF Core. На рис. 6.9 показана ситуация после выполнения строки `context.Add(book)`. Из-

менения выделены жирным шрифтом и жирными линиями для обозначения добавленных навигационных связей.

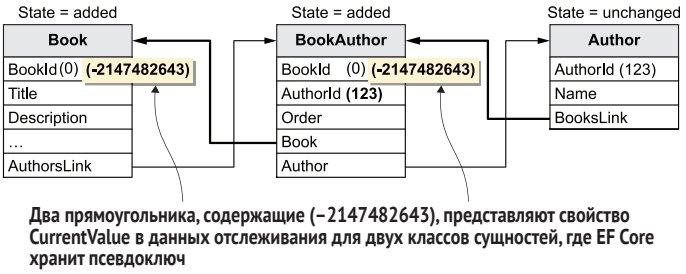


Рис. 6.9 Конец этапа 2. Здесь много чего произошло. Значение свойства `State` двух новых сущностей, `Book` и `BookAuthor`, изменилось на `Added`. В то же время метод `Add` пытается настроить внешние ключи: ему известен первичный ключ `Author`, поэтому он может установить `AuthorId` в сущности `BookAuthor`. Ему неизвестен первичный ключ `Book` (`BookId`), поэтому он помещает уникальное отрицательное число в скрытые отслеживаемые значения, действующие как псевдоключ. В методе `Add` также есть этап ссылочной фиксации, заполняющий все остальные навигационные свойства

Вы, возможно, удивлены тем, сколько всего произошло при выполнении метода `Add`. (Я – да!) Похоже, что сущности максимально приближены к тем состояниям, в которых они будут находиться после вызова метода `SaveChanges`. Вот что происходит при вызове метода `Add` на этапе 2.

Метод `Add` задает значение `Added` для State сущности, предоставленной в качестве параметра, – в данном примере это сущность `Book`. Затем он просматривает все сущности, связанные либо по навигационным свойствам, либо по значениям внешнего ключа. Для каждой связанной сущности он выполняет следующие действия:

- если сущность не отслеживается, т. е. ее текущее свойство State – `Detached`, он задает значение `Added`. В данном примере это `BookAuthor`. Для `Author` свойство State не обновляется, поскольку эта сущность отслеживается;
- он заполняет все внешние ключи правильными первичными ключами. Если связанный первичный ключ еще не доступен, он помещает уникальное отрицательное число в свойства `CurrentValue` отслеживаемых данных для первичного и внешнего ключей, как показано на рис. 6.9;
- он заполняет все навигационные свойства, которые в настоящее время не настроены, путем запуска ссылочной фиксации, описанной в разделе 6.1.1. На рис. 6.9 связи показаны жирными линиями.

В этом примере единственные сущности для привязки задает ваш код, но этап ссылочной фиксации метода `Add` может выполнить привязку к любой отслеживаемой сущности. Вызов метода `Add` может

занять некоторое время, если у вас много связей и/или много отслеживаемых классов сущностей в текущем DbContext. Я подробно рассмотрю эту проблему производительности в главе 14.

Последний этап, этап 3, – это то, что происходит при вызове метода `SaveChanges`, как показано на рис. 6.10.

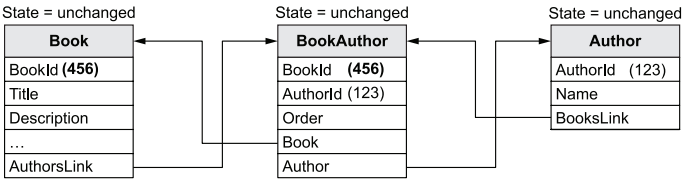


Рис. 6.10 Конец этапа 3. После завершения работы метода `SaveChanges` сущности `Book` и `BookAuthor` добавляются в базу: две новые строки были вставлены в таблицы Books и BookAuthors. Создание строки `Book` означает, что ее первичный ключ генерируется базой данных, который копируется обратно в `BookId`, а также во внешний ключ `BookAuthor`, `BookId`. По возвращении свойство `State` у `Book` и `BookAuthor` имеет значение `Unchanged`

В разделе 6.2.1 было показано, что любые столбцы, установленные или измененные базой данных, копируются обратно в класс сущности, чтобы сущность соответствовала базе данных. В этом примере были обновлены `BookId` сущности `Book` и `BookId` сущности `BookAuthor`, чтобы соответствовать значению ключа, созданному в базе данных. Кроме того, теперь, когда все сущности, участвующие в этой записи в базу данных, соответствуют базе данных, для их состояний установлено значение `Unchanged`.

Данный пример, возможно, показался длинным объяснением того, что «просто работает», и во многих случаях вам не нужно знать, почему. Но когда что-то работает неправильно, эта информация полезна, если вы хотите сделать что-то сложное, например регистрировать изменения класса сущности.

Что победит, если они разные: навигационные связи или значения внешнего ключа?

На этапе 2 в разделе 6.2.2 я отмечал, что метод `Add` «просматривает все сущности, связанные с сущностью, предоставленной в качестве параметра, либо по навигационным свойствам, либо по значениям внешнего ключа». Что победит, если навигационная связь будет привязана к одной сущности, а внешний ключ – к другой? Мои тесты говорят, что выигрывает навигационная связь. Но этот результат не определен в документации EF Core. Я попросил разъяснений (https://github.com/dotnet/efcore/issues/21105), но пока не будет ответа на этот вопрос, нужно тестировать свой код, чтобы убедиться, что в победе навигационных свойств над значениями внешнего ключа ничего не изменилось.

6.2.3 Быстрый способ копирования данных со связями

Иногда нужно скопировать класс сущности со всеми его связями. Одному из моих клиентов требовались разные версии специально разработанной структуры для отправки заказчику, чтобы он мог выбрать понравившуюся. У этих проектов было много общих частей, и проектировщики не хотели вводить эти данные для каждого варианта проектирования; они хотели создать первый вариант и скопировать его в качестве отправной точки для следующего варианта.

Можно было бы клонировать каждый класс сущности и его связи, но это тяжелая работа. (У моего клиента могли быть сотни элементов, у каждого из которых ~ 25 связей.) Но знание того, как работает EF Core, позволило мне написать код для копирования проекта с помощью самого EF Core.

В качестве примера мы будем использовать наши знания EF Core, чтобы скопировать пользовательский заказ из приложения Book App, где есть коллекция `LineItems`, которая, в свою очередь, связана с `Books`. Нам нужно скопировать `Order` только с `LineItems`, но мы не хотим копировать `Books`, с которыми связана коллекция `LineItems`; две копии `Book` вызовут проблемы. Начнем с рассмотрения заказа, который мы хотим скопировать. Он показан в следующем листинге.

Листинг 6.13 Создание `Order` с двумя объектами `LineItem`, готовыми к копированию

```
Для этого теста добавлены четыре книги, которые
будут использоваться в качестве тестовых данных
                                                    Создает Order с двумя LineItem
var books = context.SeedDatabaseFourBooks();        для копирования
var order = new Order
{
    CustomerId = Guid.Empty,                        Задает для CustomerId значение
    LineItems = new List<LineItem>                  по умолчанию, чтобы фильтр запроса
    {                                               считывал заказ обратно
        new LineItem
        {
            LineNum = 1, ChosenBook = books[0], NumBooks = 1
        },
        new LineItem
        {
            LineNum = 2, ChosenBook = books[1], NumBooks = 2
        },
    }
};
context.Add(order);
context.SaveChanges();
```

Добавляет первый LineNum, связанный с первой книгой

Добавляет второй LineNum, связанный со второй книгой

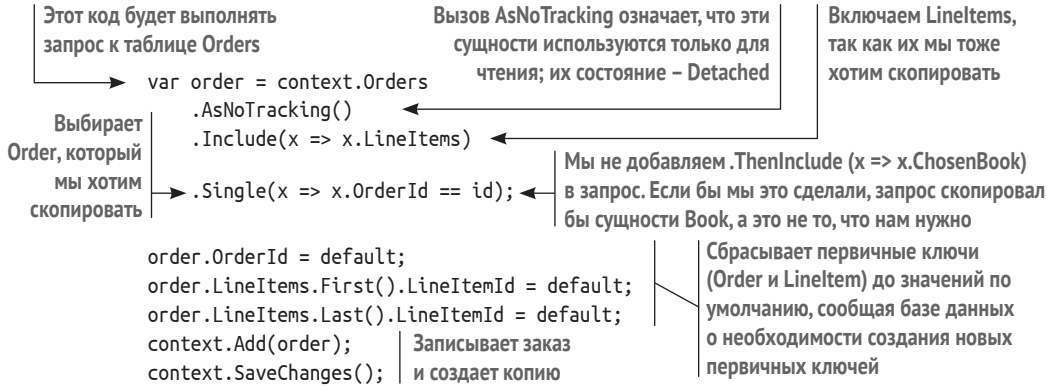
Записывает этот Order в базу данных

Чтобы правильно скопировать этот заказ, нужно знать три вещи (первые две вы уже знаете из раздела 6.2.2):

- если вы добавляете сущность, у которой есть связанные сущности, и они не отслеживаются, то есть значение свойства `State` – это `Detached`, для них будет установлено значение `Added`;
- EF Core может находить связанные сущности с помощью навигационных связей;
- если вы попытаетесь добавить класс сущности в базу данных, а первичный ключ уже находится в базе, вы получите исключение, потому что первичный ключ должен быть уникальным.

Зная эти три вещи, можно заставить EF Core скопировать `Order` с `LineItems`, но не `Books`, с которыми связаны `LineItems`. Вот код, который копирует `Order` и `LineItems`, но не копирует `Book`, связанную с `LineItems`.

Листинг 6.14 Копирование `Order` и его `LineItems`



Обратите внимание, что мы не сбрасывали внешние ключи, потому что полагаемся на тот факт, что навигационные свойства переопределяют все значения внешнего ключа. (См. предыдущую врезку «Что победит, если они разные: навигационные ссылки или значения внешнего ключа?».) Но поскольку мы действуем осторожно, то нужно создать модульный тест, чтобы проверить, скопированы ли связи должным образом.

6.2.4 Быстрый способ удалить сущность

Теперь вы можете скопировать сущность с ее связями. Как насчет быстрого удаления сущности? Оказывается, есть быстрый способ удалить ее, который хорошо подходит для удаления с отключенным состоянием, когда вы работаете с веб-приложением.

В главе 3 рассматривается удаление путем считывания сущности, которую вы хотите удалить, и последующего вызова метода EF Core `Remove` с экземпляром этой сущности. Это рабочий подход, но он требует двух обращений к базе данных – одного для чтения сущно-

сти, которую вы хотите удалить, а другого, когда вызывается метод `SaveChanges` для ее удаления. Однако, как оказалось, все, что нужно методу `Remove`, – это соответствующий класс сущности с заданным первичным ключом (ключами). В следующем листинге показано удаление сущности `Book` путем предоставления значения ее первичного ключа `BookId`.

Листинг 6.15 Удаляем сущность из базы данных, задав первичный ключ

```
var book = new Book                    // Создает класс сущности, который вы хотите удалить (в данном случае Book)
{
    BookId = bookId                    // Задает первичный ключ экземпляра сущности
};
context.Remove(book);                  // Вызов метода Remove сообщает EF Core, что вы хотите удалить эту сущность/строку
context.SaveChanges();                 // Метод SaveChanges отправляет команду в базу данных для удаления этой строки
```

В отключенном состоянии, например в каком-нибудь веб-приложении, команда на удаление возвращает только тип и значение(я) первичного ключа, что упрощает и ускоряет выполнение кода удаления. Есть некоторые мелкие отличия от подхода с чтением и удалением, применяемого к связям:

- если строки с указанным вами первичным ключом не существует, EF Core генерирует исключение `DbUpdateConcurrencyException`, сообщая, что ничего не удалено;
- база данных управляет удалением других связанных сущностей; EF Core не имеет здесь права голоса (см. обсуждение `OnDelete` в главе 8 для получения дополнительной информации).

Резюме

- При чтении классов сущностей как отслеживаемых EF Core использует процесс, называемый ссылочной фиксацией, который настраивает все навигационные свойства для любых других отслеживаемых сущностей.
- Обычный запрос с отслеживанием использует разрешение идентичности, обеспечивая наилучшее представление структуры базы данных с одним экземпляром класса сущности для каждого уникального первичного ключа.
- Запрос с методом `AsNoTracking` выполняется быстрее, нежели обычный запрос с отслеживанием, поскольку он не использует разрешение идентичности, но может создавать повторяющиеся классы сущностей с одинаковыми данными.

- Если ваш запрос загружает несколько коллекций связей с помощью метода `Include`, он создает один большой запрос к базе данных, который в некоторых случаях может выполняться медленно.
- Если в запросе пропущен метод `Include`, вы получите неправильный результат, но есть способ настроить свои навигационные коллекции таким образом, чтобы ваш код завершался ошибкой, вместо того чтобы возвращать неверные данные.
- Использование глобальных фильтров запросов для реализации функции мягкого удаления прекрасно работает, но следите, как вы обрабатываете связи, которые зависят от мягко удаленной сущности.
- Выборочные запросы эффективны со стороны базы данных, но для них приходится писать больше кода. Библиотека AutoMapper может автоматизировать создание таких запросов.
- EF Core создает класс сущности при чтении данных. Он делает это с помощью конструктора без параметров по умолчанию или любых других конструкторов, которые вы пишете, если следуете паттерну «По соглашению».
- Когда EF Core создает сущность в базе данных, он считывает любые данные, сгенерированные базой данных, например первичный ключ, предоставленный базой, чтобы можно было обновить экземпляр класса сущности в соответствии с базой данных.

Часть II

Об Entity Framework в деталях

В первой части было показано, как создать приложение с помощью EF Core. Во второй части рассказывается, как настроить EF Core именно так, как вам нужно, и различные способы изменения (говоря терминами EF Core: *миграции*) базы данных. Кроме того, она знакомит вас с расширенными функциями, которые могут сделать ваше программное обеспечение более эффективным с точки зрения разработки и производительности. Это скорее справочный раздел, подробно описывающий каждую часть EF Core, но (надеюсь) делая это увлекательным способом.

Глава 7 знакомит вас с тем, как EF Core настраивается, когда используется впервые, чтобы вы знали, где и как применять собственные конфигурации. В этой главе основное внимание уделяется нереляционным свойствам следующих типов: `int`, `string` и `DateTime`.

В главе 8 показано, как EF Core находит и конфигурирует связи. Он отлично справляется с настройкой большинства связей, но в некоторых случаях нуждается в помощи, когда связи придется настраивать вручную, потому что настройки EF Core по умолчанию могут не соответствовать вашим потребностям.

В главе 9 рассматривается важный вопрос сопоставления базы данных с вашей конфигурацией EF Core с использованием программного обеспечения или базы данных SQL. В ней описаны различные способы безопасного изменения – миграции – базы данных по мере развития приложения.

В главе 10 рассматриваются более сложные настраиваемые функции, такие как определение вычисляемых столбцов в базе данных, а также отслеживание и обработка параллельных обновлений базы данных. Вы будете использовать эти функции только в определенных обстоятельствах, но должны знать, что они существуют на случай, если они вам понадобятся.

В главе 11 рассматриваются методы внутри класса DbContext, особенно то, как метод `SaveChanges` решает, что записывать в базу данных и как можно повлиять на это. В ней описываются и другие темы, такие как доступ к базе данных с использованием чистого SQL, отказоустойчивость подключения к базе данных и свойство `DbContext.Model`.

7 Настройка нереляционных свойств

В этой главе рассматриваются следующие темы:

- три способа настройки EF Core;
- фокусируемся на нереляционных свойствах;
- определение структуры базы данных;
- знакомство с преобразователями значений, теневыми свойствами и резервными полями;
- решаем, какой тип конфигурации лучше всего подходит для разных ситуаций.

В этой главе рассказывается о настройке EF Core в целом, но основное внимание уделяется настройке нереляционных свойств в классе сущности; эти свойства известны как *скалярные*. Глава 8 посвящена настройке реляционных свойств, а в главе 10 идет речь о настройке более продвинутых функций, таких как DbFunctions, вычисляемые столбцы и др.

Эта глава начинается с обзора процесса настройки, который EF Core запускает при первом использовании DbContext приложения. После этого вы узнаете, как настроить отображение между классами .NET и связанными с ними таблицами базы данных, используя такие функции, как установка имени, типа SQL и допустимости значений NULL для столбцов в таблице.

В этой главе также представлены три функции EF Core: *преобразователи значений*, *теневые свойства* и *резервные поля*, – позволяю-

щие контролировать, как данные хранятся и управляются остальной частью кода, отличного от EF Core. Преобразователи значений, например, позволяют преобразовывать данные при записи или чтении их из базы данных, что упрощает понимание и отладку представления базы данных; теневые свойства и резервные поля позволяют «скрывать» или контролировать доступ к данным базы данных на программном уровне. Эти возможности могут помочь вам писать более качественные и менее уязвимые приложения, которые легче отлаживать и выполнять рефакторинг.

7.1 Три способа настройки EF Core

В главе 1 было рассмотрено, как EF Core моделирует базу данных с иллюстрацией того, что он делает, с точки зрения базы данных. На рис. 7.1 более подробно показан процесс настройки, который происходит при первом использовании DbContext. На нем изображен весь процесс с тремя подходами: «По соглашению», Data Annotations и Fluent API. В этом примере основное внимание уделяется настройке скалярных свойств, но процесс одинаковый для всех конфигураций EF Core.

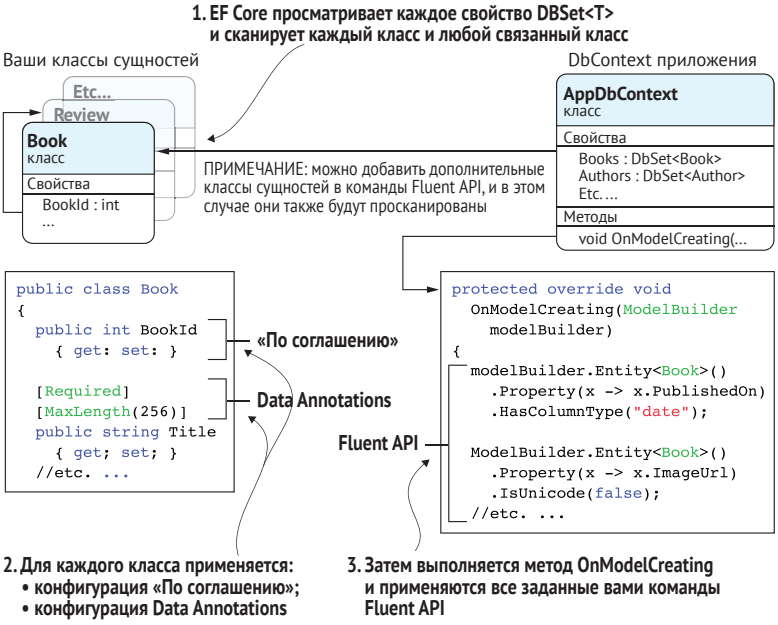


Рис. 7.1 При первом использовании DbContext EF Core запускает процесс настройки и построения модели базы данных, к которой он должен получить доступ. Для настройки EF Core можно использовать три подхода: «По соглашению», Data Annotations и Fluent API. Большинству реальных приложений необходимо сочетание всех трех подходов для настройки EF Core в точности так, как требуется вашему приложению

В этом списке перечислены три подхода к настройке EF Core:

- «*По соглашению*» – когда вы следуете простым правилам для типов и имен свойств, EF Core автоматически настраивает многие функции программного обеспечения и базы данных. Это быстрый и простой подход, но он не может справиться со всеми непредвиденными обстоятельствами;
- *Data Annotations* – ряд атрибутов .NET, известных как аннотации данных, можно добавить к классам сущностей и/или свойствам для предоставления дополнительной информации о конфигурации. Эти атрибуты также могут быть полезны для проверки данных, описанной в главе 4;
- *Fluent API* – в EF Core есть метод `OnModelCreating`, который выполняется при первом использовании контекста EF. Можно переопределить этот метод, а также добавить ряд команд, известных как *Fluent API*, чтобы предоставить EF Core дополнительную информацию на этапе моделирования. Fluent API – это наиболее полная форма информации о конфигурации, а некоторые функции доступны только через этот API.

ПРИМЕЧАНИЕ Большинство реальных приложений должны использовать все три подхода к настройке EF Core и базы данных именно так, как им нужно. Некоторые возможности доступны при использовании двух или даже всех трех подходов (например, определение первичного ключа в классе сущности). В разделе 7.16 я привожу рекомендации относительно того, какой подход использовать для определенных функций, а также способ автоматизации некоторых конфигураций.

7.2 Рабочий пример настройки EF Core

Для приложений более высокого уровня сложности, нежели Hello-World-программы, вам, вероятно, понадобится некая разновидность Аннотаций данных или Fluent API. В части I нам нужно было настроить ключ для таблицы связей «многие ко многим». В этой главе вы увидите пример применения всех трех подходов, представленных в разделе 7.1, чтобы база данных лучше соответствовала потребностям нашего приложения Book App.

В этом примере мы модифицируем класс сущности `Book`, используемый в главах 2–5, изменив значения по умолчанию (задаваемые EF Core) для размера и типа некоторых столбцов посредством миграции. Эти изменения уменьшат размер базы данных, ускорят сортировку и поиск по некоторым столбцам и проверят, не содержат ли какие-либо столбцы значения `null`. Считается хорошей практикой задавать отвечающие бизнес-потребностям размер, тип и допустимость этих значений для столбцов базы данных.

Для этого мы будем использовать сочетание всех трех подходов. Конфигурация «По соглашению» играет важную роль, поскольку определяет имена таблиц и столбцов, но мы добавим специальные аннотации данных и методы Fluent API, чтобы изменить некоторые столбцы. На рис. 7.2 показано, как каждый подход влияет на внутреннюю модель структуры таблиц базы данных EF Core. Из-за нехватки места на рисунке не показаны все аннотации данных и конфигурационные методы Fluent API, примененные к таблице, но их можно увидеть в листингах 7.1 и 7.2.

ПРИМЕЧАНИЕ На рис. 7.2 стрелки используются для связывания кода различных конфигураций EF Core с частями столбцов таблицы базы данных. Абсолютно очевидно, что изменение конфигурации EF Core не ведет к магическому изменению базы данных. В главе 9, посвященной изменению структуры базы данных (известной как схема), описывается несколько способов, с помощью которых конфигурации EF Core изменяют базу данных или база данных изменяет конфигурации EF Core в вашем коде.

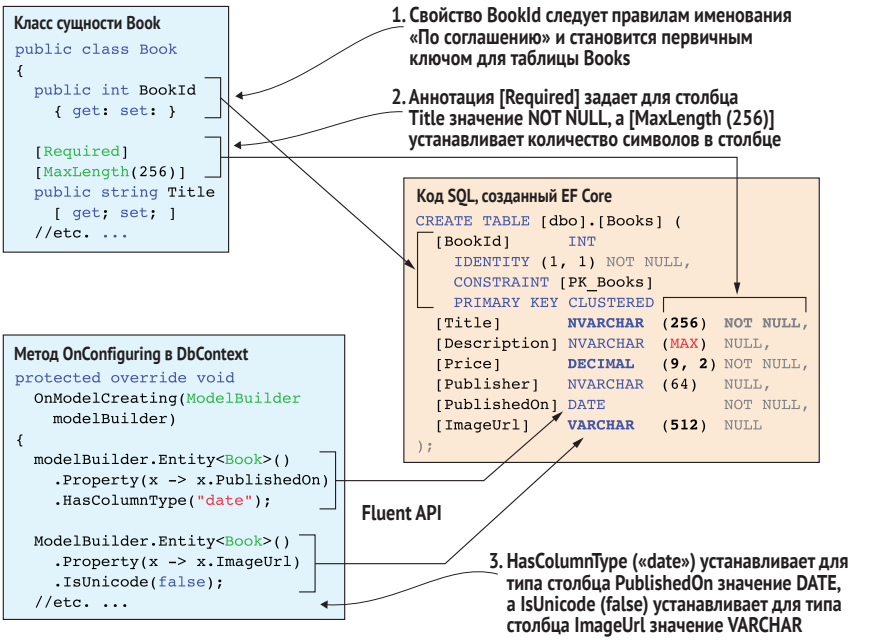


Рис. 7.2 Чтобы задать нужный формат таблице Books, необходимо использовать все три подхода. Бóльшая часть выполнена с использованием подхода «По соглашению» (все, что оформлено обычным, нежирным шрифтом), затем используются аннотации данных – чтобы установить размер и допустимость пустых значений столбца Title, и Fluent API – чтобы изменить тип столбцов `PublishedOn` и `ImageUrl`

Вы увидите более подробные объяснения этих настроек по ходу чтения главы, но в этой части дается общее представление о различных способах настройки DbContext. Кроме того, интересно подумать о том, чем некоторые из этих конфигураций могут быть полезны в ваших собственных проектах. Вот несколько настроек EF Core, которые я использую в большинстве проектов, над которыми работаю:

- *атрибут* `[Required]` – примененный к свойству Title сообщит EF Core, что соответствующий столбец не может иметь значение NULL, а это означает, что база данных вернет ошибку, если вы попытаетесь вставить или обновить книгу со значением `Title`, равным NULL;
- *атрибут* `[MaxLength(256)]` – сообщает EF Core, что количество символов, хранящихся в базе данных, должно составлять 256, а не максимальное значение по умолчанию для базы данных (2 ГБ в SQL Server). Наличие строк фиксированной длины правильного типа, 2-байтового Unicode или 1-байтового ASCII, делает доступ к базе данных немного эффективнее и позволяет применять индекс SQL к этим столбцам фиксированного размера;

ОПРЕДЕЛЕНИЕ *Индекс SQL* – это функция, повышающая производительность сортировки и поиска. Эта тема более подробно освещена в разделе 7.10.

- `HasColumnType("date")` – если сделать так, чтобы столбец `PublishedOn` содержал только дату (а это все, что вам нужно), а не `datetime2` по умолчанию, то вы уменьшите размер столбца с 8 до 3 байт, что ускорит поиск и сортировку по столбцу `PublishedOn`;
- `IsUnicode(false)` – свойство `ImageUrl` содержит только 8-битные символы в кодировке ASCII, поэтому вы сообщаете об этом EF Core, а это означает, что строка будет сохранена именно таким образом. Итак, если у свойства `ImageUrl` есть атрибут `[MaxLength(512)]` (как показано в листинге 7.1), метод `IsUnicode(false)` уменьшит размер столбца `ImageUrl` с 1024 (Юникод занимает 2 байта на символ) до 512 байт (ASCII занимает 1 байт на символ).

В этом листинге показан обновленный код класса сущности `Book` с новыми аннотациями данных, выделенными жирным шрифтом. (Методы Fluent API описаны в разделе 7.5.)

Листинг 7.1 Класс сущности Book с добавленными аннотациями данных

```
public class Book
{
    public int BookId { get; set; }

    [Required]
```

Сообщает EF Core, что строка не допускает значения NULL

```
    [MaxLength(256)]
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime PublishedOn { get; set; }
    [MaxLength(64)]
    public string Publisher { get; set; }
    public decimal Price { get; set; }

    [MaxLength(512)]
    public string ImageUrl { get; set; }
    public bool SoftDeleted { get; set; }

    //------------------------------------------------
    //Связи;

    public PriceOffer Promotion { get; set; }
    public IList<Review> Reviews { get; set; }
    public IList<BookAuthor> AuthorsLink { get; set; }
}
```

СОВЕТ Обычно параметр размера задается в атрибуте `[MaxLength(nn)]` с использованием константы, поэтому если вы создаете DTO – используйте ту же самую константу. Если вы измените размер одного свойства, то измените все связанные свойства.

Теперь, когда вы ознакомились с примером, в котором применяются все три подхода к настройке, подробно изучим каждый подход.

7.3 Конфигурация по соглашению

«*По соглашению*» – это конфигурация по умолчанию, которую можно переопределить двумя другими подходами: аннотациями данных и Fluent API. Подход «*По соглашению*» полагается на разработчика, который будет использовать стандарты именования и отображения типов, позволяющие EF Core находить и настраивать классы сущностей и их связи, а также определять бóльшую часть модели базы данных. Такой подход обеспечивает быстрый способ настройки большой части отображения в базу данных, поэтому его стоит освоить.

7.3.1 Соглашения для классов сущностей

Классы, которые EF Core отображает в базу данных, называются *классами сущностей*. Как указано в главе 2, классы сущностей – это обычные классы .NET, иногда называемые POCO (Plain Old CLR Object). EF Core требует, чтобы классы сущностей обладали следующими особенностями:

- класс должен быть открытым – перед классом должно стоять ключевое слово `public`;

- класс не может быть статическим, поскольку EF Core должен иметь возможность создавать новый экземпляр класса;
- у класса должен быть конструктор, который может использовать EF Core. Подойдет как конструктор по умолчанию без параметров, так и конструкторы с параметрами. Подробные правила использования конструкторов в EF Core см. в разделе 6.1.10.

7.3.2 Соглашения для параметров в классе сущности

По соглашению EF Core будет искать открытые свойства в классе сущности, у которых есть открытые методы чтения и записи с любым модификатором доступа (`public`, `internal`, `protected` или `private`). Типичное открытое для всех свойство выглядит так:

```
public int MyProp { get; set; }
```

Хотя открытое для всех свойство является нормой, в некоторых ситуациях свойство с более ограниченным доступом (например, `public int MyProp {get; private set;}`) позволяет лучше контролировать его настройку. Один из примеров – метод в классе сущности, который также выполняет проверки перед установкой свойства; см. главу 13 для получения дополнительной информации.

ПРИМЕЧАНИЕ EF Core может обрабатывать свойства с доступом только на чтение – свойства только с методом чтения, например `public int MyProp {get; }`. Но в этом случае подход «По соглашению» не сработает; нужно использовать Fluent API, чтобы сообщить EF Core, что эти свойства отображаются в базу данных.

7.3.3 Условные обозначения для имени, типа и размера

Вот правила обозначения имени, типа и размера столбца реляционной базы данных:

- в качестве имени столбца в таблице используется имя свойства класса сущности;
- тип .NET преобразуется поставщиком базы данных в соответствующий SQL-тип. Многие базовые типы .NET отображаются в соответствующие типы базы данных по принципу «один к одному». Эти базовые типы в основном являются *примитивными* типами .NET (`int`, `bool` и т. д.) с некоторыми особыми случаями (такими как `string`, `DateTime` и `Guid`);
- размер определяется типом .NET, например 32-битный тип `int` хранится в соответствующем 32-битном типе SQL, INT. Типы `string` и `byte[]` принимают максимальный размер, который будет отличаться для каждого типа базы данных.

EF6 Одно из изменений в соглашениях об отображении по умолчанию заключается в том, что EF Core отображает тип .NET `DateTime` в тип SQL `datetime2(7)`, тогда как EF6 отображает его в `datetime`. Microsoft рекомендует использовать `datetime2(7)`, потому что он следует стандарту ANSI и ISO SQL. Кроме того, `datetime2(7)` более точный: точность `datetime` составляет около 0,004 секунды, тогда как `datetime2(7)` имеет точность 100 наносекунд.

7.3.4 По соглашению поддержка значения NULL для свойства основана на типе .NET

В реляционных базах данных `NULL` обозначает отсутствующее или неизвестное значение. Может ли столбец иметь значение `NULL` – определяется типом .NET:

- если тип – `string`, то столбец может иметь значение `NULL`, поскольку строка может иметь значение `NULL`;
- примитивные типы (например, `int`) или структуры (например, `DateTime`) по умолчанию не имеют значения `NULL`;
- примитивные типы или структуры можно сделать таковыми с помощью суффикса `?` (например, `int?`) или обобщенного класса `Nullable<T>` (например, `Nullable<int>`). В этих случаях столбец может иметь значение `NULL`.

На рис. 7.3 показаны соглашения об имени, типе, размере и допустимости значений NULL, применяемые к свойству.

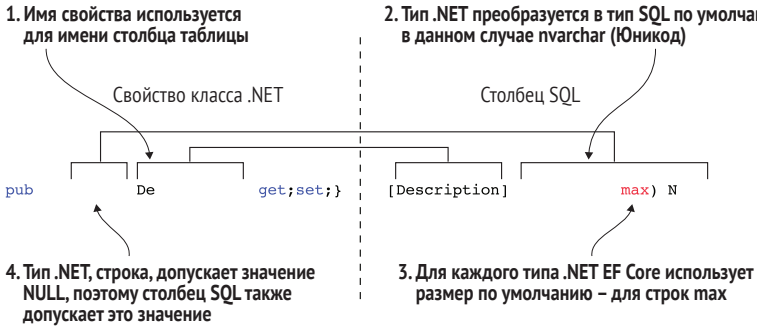


Рис. 7.3 Применение правил «*По соглашению*» для определения столбца SQL. Тип свойства преобразуется поставщиком базы данных в эквивалентный тип SQL, тогда как имя свойства используется для имени столбца

7.3.5 Соглашение об именах EF Core определяет первичные ключи

Другое правило касается определения первичного ключа таблицы базы данных. Соглашения EF Core для обозначения первичного ключа выглядят так:

- EF Core ожидает одно свойство в качестве первичного ключа (Подход «По соглашению» не обрабатывает ключи, состоящие из нескольких свойств или столбцов, которые называются *составными ключами*.);
- свойство называется `Id` или `<имя класса>id` (например, `BookId`);
- тип свойства определяет, кто отвечает за генерацию уникальных значений ключа. О генерации ключей рассказывается в главе 8.

На рис. 7.4 показан пример первичного ключа, сгенерированного базой данных, с отображением «По соглашению» для свойства `BookId` и столбца BookId таблицы Books.

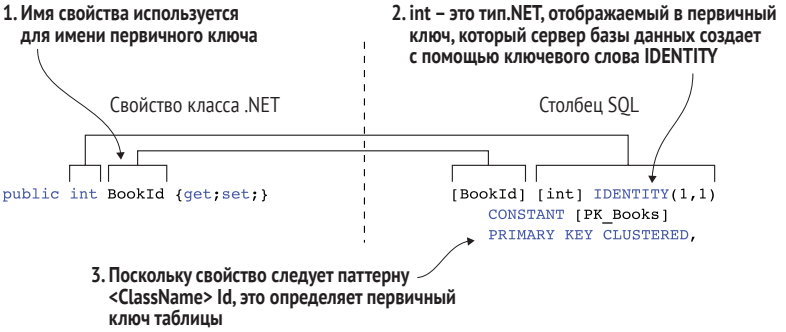


Рис. 7.4 Отображение свойства `BookId` класса .NET в первичный столбец BookId с использованием подхода «По соглашению». Имя свойства сообщает EF Core, что данное свойство – это первичный ключ. Кроме того, поставщик базы данных знает: тип `int` означает, что он должен создать уникальное значение для каждой строки, добавленной в таблицу

СОВЕТ Хотя есть возможность использовать короткое имя `Id` для первичного ключа, я рекомендую использовать более длинное имя `<имя класса>`, за которым следует `Id` (например, `BookId`). Понять, что происходит в вашем коде, проще, если использовать `Where(p => BookId == 1)` вместо более короткого `Where(p => Id == 1)`, особенно когда у вас много классов сущностей.

7.4 Настройка с помощью аннотаций данных

Аннотации данных – это особый тип атрибутов .NET, используемый для валидации и использования функциональных возможностей базы данных. Эти атрибуты могут применяться к классу сущности или свойству и предоставлять EF Core информацию о конфигурации. В этом разделе рассказывается, где их найти и как они обычно применяются. Атрибуты аннотаций данных, относящиеся к конфигурации EF Core, берут свое начало из двух пространств имен.

7.4.1 Использование аннотаций из пространства имен System.ComponentModel.DataAnnotations

В основном атрибуты из пространства имен `System.ComponentModel.DataAnnotations` используются для валидации данных в клиентской части, например ASP.NET, но EF Core использует некоторые из них для создания модели отображения. Такие атрибуты, как `[Required]` и `[MaxLength]`, являются основными, при этом многие другие аннотации данных не влияют на EF Core. На рис. 7.5 показано, как основные атрибуты `[Required]` и `[MaxLength]` влияют на определение столбцов базы данных.

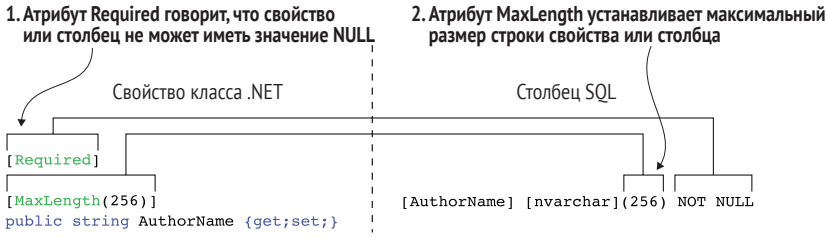


Рис. 7.5 Атрибуты `[Required]` и `[MaxLength]` влияют на отображение в столбец базы данных. Атрибут `[Required]` указывает на то, что столбец не может иметь значение NULL, а атрибут `[MaxLength]` задает максимальную длину `nvarchar`

7.4.2 Использование аннотаций из пространства имен System.ComponentModel.DataAnnotations.Schema

Атрибуты из пространства имен `System.ComponentModel.DataAnnotations.Schema` более специфичны для конфигурации базы данных. Это пространство имен было добавлено в NET Framework 4.5 задолго до того, как был написан EF Core, но EF Core использует его атрибуты, такие как `[Table]`, `[Column]` и т. д., чтобы задать имя таблицы и имя и тип столбца, как описано в разделе 7.11.

7.5 Настройка с использованием Fluent API

Третий подход к настройке EF Core под названием *Fluent API* представляет собой набор методов, которые работают с классом `ModelBuilder`. Он доступен в методе `OnModelCreating` внутри DbContext. Как вы увидите, Fluent API работает с помощью методов расширения, которые можно объединить в цепочку, как команды LINQ, чтобы задать параметр конфигурации. Fluent API предоставляет наиболее полный список методов конфигурирования, многие из которых доступны только через этот API.

Но прежде чем демонстрировать эти методы, я хочу представить другой подход, который разделяет методы Fluent API на группы по классам каждой сущности. Это полезный подход, потому что по мере роста приложения размещение всех методов Fluent API в методе `OnModelCreating` (как показано на рис. 2.6) усложняет поиск конкретного Fluent API. Решение – переместить Fluent API для класса сущности в отдельный класс конфигурации, который затем вызывается из метода `OnModelCreating`.

EF Core предоставляет метод для облегчения этого процесса в виде интерфейса `IEntityTypeConfiguration<T>`. В листинге 7.2 показан наш новый DbContext приложения, `EfCoreContext`, где мы перемещаем настройку Fluent API для различных классов в отдельные классы конфигурации. Преимущество этого подхода состоит в том, что Fluent API для класса сущности находится в одном месте и не смешивается с командами Fluent API для других классов сущностей.

EF6 В EF6.x есть класс `EntityTypeConfiguration<T>`, который можно наследовать, чтобы инкапсулировать конфигурацию Fluent API для данного класса сущности. Реализация EF Core дает тот же результат, но использует интерфейс `IEntityTypeConfiguration<T>`, который применяется к классу конфигурации.

Листинг 7.2 DbContext приложения для базы данных со связями

```
public class EfCoreContext : DbContext
{
    public EfCoreContext(DbContextOptions<EfCoreContext> options)
        : base(options)
    { }

    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }
    public DbSet<PriceOffer> PriceOffers { get; set; }
    public DbSet<Order> Orders { get; set; }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new BookConfig());
        modelBuilder.ApplyConfiguration(new BookAuthorConfig());
        modelBuilder.ApplyConfiguration(new PriceOfferConfig());
        modelBuilder.ApplyConfiguration(new LineItemConfig());
    }
}
```

UseId пользователя, купившего книги

Создает DbContext, используя параметры, установленные при регистрации

Классы сущностей, к которым будет обращаться ваш код

Метод, в котором выполняются методы Fluent API

Запускаем каждую отдельную конфигурацию для каждого класса сущности, который требует настройки

Посмотрим на класс `BookConfig`, использованный в листинге 7.2, чтобы увидеть, как создать класс конфигурации для каждого типа. В листинге 7.3 показан класс конфигурации, реализующий интерфейс `IEntityTypeConfiguration<T>` и содержащий методы Fluent API для класса сущности `Book`.

ПРИМЕЧАНИЕ Я не описываю Fluent API из листинга 7.3, потому что это пример использования интерфейса `IEntityTypeConfiguration<T>`. Fluent API рассматриваются в разделах 7.7 (тип базы данных) и 7.10 (индексы).

Листинг 7.3 Класс конфигурации `BookConfig` настраивает класс сущности `Book`

```
internal class BookConfig : IEntityTypeConfiguration<Book>
{
    public void Configure
        (EntityTypeBuilder<Book> entity)
    {
        entity.Property(p => p.PublishedOn)
            .HasColumnType("date");

        entity.Property(p => p.Price)
            . HasPrecision(9,2);

        entity.Property(x => x.ImageUrl)
            .IsUnicode(false);

        entity.HasIndex(x => x.PublishedOn);
    }
}
```

Основанное на соглашении отображение для DateTime в .NET – это datetime2 в SQL. Эта команда изменяет тип столбца SQL на date, который содержит только дату, но не содержит время

Точность (9,2) устанавливает максимальную цену 9 999 999,99 (9 цифр, две после десятичного разделителя), которая занимает наименьший размер в базе данных

Основанное на соглашении отображение для строки .NET – это nvarchar в SQL (16-битный Unicode). Эта команда изменяет тип столбца SQL на varchar (8-битный ASCII)

Добавляет индекс к свойству PublishedOn, потому что по этому свойству выполняются сортировка и фильтрация

В листинге 7.2 я перечисляю каждый из отдельных вызовов `modelBuilder.ApplyConfiguration`, чтобы увидеть их в действии. Но экономящий время метод `ApplyConfigurationsFromAssembly` может найти все ваши классы конфигурации, которые наследуются от `IEntityTypeConfiguration<T>`, и запустить их все за вас. Посмотрите на следующий фрагмент кода, который находит и запускает все ваши классы конфигурации в той же сборке, что и DbContext:

```
modelBuilder.ApplyConfigurationsFromAssembly(
     Assembly.GetExecutingAssembly());
```

В листинге 7.3 показано типичное использование Fluent API, но помните, что характер этого API позволяет объединять несколько команд в цепочку, как показано в этом фрагменте кода:

```
modelBuilder.Entity<Book>()
    .Property(x => x.ImageUrl)
    .IsUnicode(false)
    .HasColumnName("DifferentName")
    .HasMaxLength(123)
    .IsRequired(false);
```

EF6 Fluent API работает так же, как и в EF6.x, но здесь у него множество нововведений, существенные изменения в настройке связей (рассматриваются в главе 8) и тонкие изменения в типах данных.

Метод `OnModelCreating` вызывается, когда приложение впервые обращается к DbContext. На этом этапе EF Core конфигурируется, используя все три подхода: «По соглашению», Data Annotations и любой Fluent API, добавленный в метод `OnModelCreating`.

Что, если Data Annotations и Fluent API говорят о разном?

Методы моделирования Data Annotations и Fluent API всегда имеют приоритет над моделированием на основе соглашений. Но что произойдет, если Data Annotations и Fluent API предоставляют отображение одного и того же свойства или настройку?

Я попытался задать для типа и длины свойства `WebUrl` разные значения через Data Annotations и Fluent API. Были использованы значения Fluent API. Этот тест не был определяющим, но логично, что окончательным арбитром был Fluent API.

Теперь, когда вы узнали о подходах Data Annotations и Fluent API, подробно рассмотрим конфигурацию отдельных частей модели базы данных.

7.6 Исключение свойств и классов из базы данных

В разделе 7.3.2 описано, как EF Core находит свойства. Но иногда нужно исключить данные в классах сущностей из базы данных. Например, вам нужны локальные данные для расчета, используемые во время жизненного цикла экземпляра класса, но вы не хотите, чтобы они сохранялись в базе данных. Класс или свойство можно исключить двумя способами: через Data Annotations или Fluent API.

7.6.1 Исключение класса или свойства с помощью Data Annotations

EF Core исключит свойство или класс, к которому применен атрибут данных `[NotMapped]`. В следующем листинге показано применение этого атрибута как к свойству, так и к классу.

Листинг 7.4 Исключаем три свойства, два с использованием атрибута [NotMapped]

```
public class MyEntityClass
{
    public int MyEntityClassId { get; set; }

    public string NormalProp{ get; set; }

    [NotMapped]
    public string LocalString { get; set; }

    public ExcludeClass LocalClass { get; set; }
}

[NotMapped]
public class ExcludeClass
{
    public int LocalInt { get; set; }
}
```

Включено: обычное открытое свойство с открытыми методами чтения и записи

Исключено: размещение атрибута [NotMapped] указывает EF Core не отображать это свойство в столбец в базе данных

Исключено: этот класс не будет включен в базу данных, потому что в определении класса есть атрибут [NotMapped]

Исключено: этот класс будет исключен, поскольку в определении класса есть атрибут [NotMapped]

7.6.2 Исключение класса или свойства с помощью Fluent API

Кроме того, можно исключить свойства и классы с помощью команды Fluent API `Ignore`, как показано в листинге 7.5.

ПРИМЕЧАНИЕ Для простоты я демонстрирую использование Fluent API внутри метода `OnModelCreating`, а не в отдельном классе конфигурации.

Листинг 7.5 Исключение свойства и класса с помощью Fluent API

```
public class ExcludeDbContext : DbContext
{
    public DbSet<MyEntityClass> MyEntities { get; set; }

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<MyEntityClass>()
            .Ignore(b => b.LocalString);

        modelBuilder.Ignore<ExcludeClass>();
    }
}
```

Метод Ignore используется для исключения свойства LocalString в классе сущности MyEntityClass, чтобы не добавлять его в базу данных

Другой метод Ignore может исключить класс таким образом, что если у вас есть свойство с игнорируемым типом, это свойство не добавляется в базу данных

Как было сказано в разделе 7.3.2, по умолчанию EF Core игнорирует свойства с доступом только на чтение, т. е. только с методом чтения (например, `public int MyProp {get; }`).

7.7 Установка типа, размера и допустимости значений NULL для столбца базы данных

Как описано ранее, моделирование на основе соглашений использует значения по умолчанию для типа, размера/точности и допустимости значений NULL на основе типа .NET. Общее требование состоит в том, чтобы установить один или несколько из этих атрибутов вручную, потому что используется существующая база данных или потому, что вы исходите из соображений производительности или бизнеса.

Во введении (раздел 7.3) мы работали с примером, где изменяли тип и размер различных столбцов. В табл. 7.1 представлен полный список команд, доступных для выполнения этой задачи.

Таблица 7.1 Настройка допустимости значений NULL и типа/размера для столбца

Параметр	Data Annotations	Fluent API

Установить значение, отличное от NULL (по умолчанию допускает значение NULL)	`[Required]`
	`public string MyProp`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>()`
		`.Property(p => p.MyProp)`
		`.IsRequired();`

Установить длину строки (по умолчанию – MAX)	`[MaxLength(123)]`
	`public string MyProp`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>()`
		`.Property(p => p.MyProp)`
		`.HasMaxLength(123);`

Установить тип/размер SQL (у каждого типа есть точность и размер по умолчанию)	`[Column(TypeName =`
	`"date")]`
	`public DateTime`
	`PublishedOn`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>(`
		`.Property(p =>`
		`p.PublishedOn)`
		`.HasColumnType("date");`

У некоторых конкретных типов SQL есть свои команды Fluent API, которые показаны в следующем списке. Их можно увидеть в листинге 7.3:

- `IsUnicode(false)` – задает для типа SQL значение `varchar(nnn)` (1-байтовый символ, известный как ASCII), а не значение по умолчанию `nvarchar(nnn)` (2-байтовый символ, известный как Юникод);
- `HasPrecision(precision, scale)` – задает количество цифр (параметр `precision`) и сколько цифр стоит после десятичного разделителя (параметр `scale`). Это новая команда Fluent API в EF Core 5. По умолчанию значение для `decimal` – (18,2);
- `HasCollation("collation name")` – еще одна функция EF Core 5, позволяющая определять параметры сопоставления, т. е. правила сортировки, чувствительность к регистру и диакритическим знакам типов `char` и `string` (см. раздел 2.8.3 для получения дополнительной информации о сопоставлении).

Я рекомендую использовать метод `IsUnicode(false)`, чтобы сообщить EF Core, что строковое свойство содержит только однобайтовые символы формата ASCII, потому что использование метода `IsUnicode` позволяет установить размер строки отдельно.

EF6 У EF Core несколько иной подход к настройке типа данных SQL столбца. Если вы указываете тип данных, нужно дать полное определение: тип и длину/точность, например `[Column (TypeName = "varchar(nnn)")]`, где `nnn` – целое число. В EF6 можно использовать `[Column(TypeName = "varchar")]`, а затем определить длину с помощью `[MaxLength (nnn)]`, но этот метод не работает в EF Core. См. https://github.com/dotnet/efcore/issues/3985 для получения дополнительной информации.

7.8 Преобразование значения: изменение данных при чтении из базы данных или записи в нее

Функция преобразования значений EF Core позволяет изменять данные при чтении и записи свойства в базу данных. Типичные варианты использования:

- сохранение `Enum`-значений в виде строки (вместо числа) для простоты восприятия при просмотре данных в базе данных;
- устранение проблемы потери типом `DateTime` настройки UTC (всемирное координированное время) при чтении из базы данных;
- (продвинутый уровень) шифрование свойства, записываемого в базу данных, и его расшифровка во время считывания.

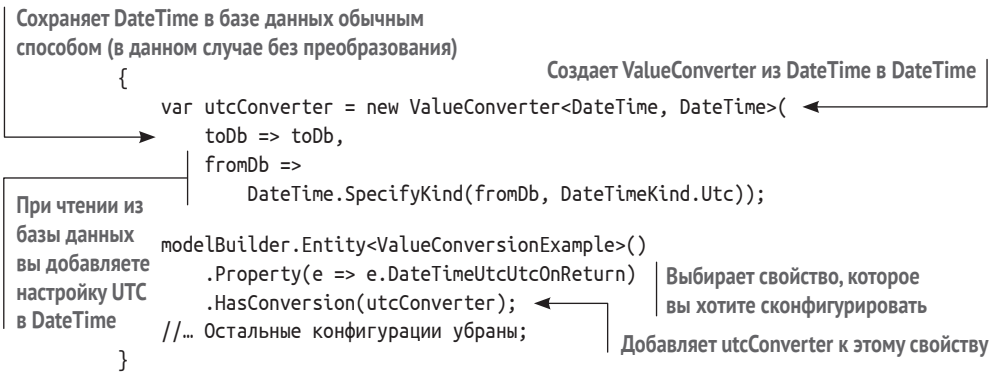
Преобразование значения состоит из двух частей:

- код, преобразующий данные во время их записи в базу данных;
- код, преобразующий столбец базы данных обратно в исходный тип при чтении.

Первый пример преобразования значений связан с ограничением базы данных SQL при хранении типов `DateTime`, поскольку он не сохраняет часть `DateTimeKind` структуры `DateTime`, которая сообщает нам, является ли `DateTime` местным или всемирным координированным временем. Эта ситуация может вызвать проблемы. Если вы отправляете `DateTime` в клиентскую часть с помощью, например, JSON, `DateTime` не будет содержать символ суффикса Z, который сообщает JavaScript, что это UTC, поэтому код клиентской части может отображать неправильное время. В следующем листинге показано, как настроить свойство для преобразования значения, которое устанавливает `DateTimeKind` при возврате из базы данных.

Листинг 7.6 Настройка свойства `DateTime` для замены потерянного параметра `DateTimeKind`

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
```




```
{
    var utcConverter = new ValueConverter<DateTime, DateTime>(
        toDb => toDb,
        fromDb =>
            DateTime.SpecifyKind(fromDb, DateTimeKind.Utc));

    modelBuilder.Entity<ValueConversionExample>()
        .Property(e => e.DateTimeUtcUtcOnReturn)
        .HasConversion(utcConverter);
    //… Остальные конфигурации убраны;
}
```

В этом случае нам пришлось создать собственный преобразователь значений, но в EF Core доступно около 20 встроенных преобразователей. (См. http://mng.bz/mgYP.) Причем один из них настолько популярен, что у него есть предопределенный метод (атрибут, если точнее) Fluent API – преобразование для хранения `Enum` в виде строки в базе данных. Позвольте мне объяснить.

Обычно `Enum` хранятся в базе данных в виде чисел. Это эффективный формат, но он усложняет задачу, если нужно заглянуть в базу данных, чтобы выяснить, что произошло. Поэтому некоторым разработчикам нравится сохранять их в базе данных в виде строки. Можно настроить преобразование `Enum` в строку с помощью команды `HasConversion<string>()`, как в следующем фрагменте кода:

```
modelBuilder.Entity<ValueConversionExample>()
    .Property(e => e.Stage)
    .HasConversion<string>();
```

Ниже приведены некоторые правила и ограничения на использование преобразования значений.

- Значение NULL не должно передаваться в преобразователь значений. Нужно написать преобразователь значений для обработки только значения, отличного от NULL, поскольку преобразователь будет вызываться лишь в том случае, если значение не равно NULL.
- Следите за запросами, содержащими сортировку по преобразованному значению. Например, если вы преобразовали `Enum` в строку, то сортировка будет выполняться по имени `Enum`, а не по его значению.
- Преобразователь может отобразить только одно свойство в один столбец в базе данных.
- Можно создать несколько сложных преобразователей значений, например сериализовать список `int` в строку JSON. В текущей версии EF Core не сможет сопоставить свойство `List<int>` с JSON в базе данных, поэтому не будет обновлять ее. Чтобы решить эту проблему, нужно добавить *компаратор значений*. Посетите стра-

ницу http://mng.bz/5j5z для получения дополнительной информации по данной теме.

Позже, в разделе 7.16.4, вы узнаете, как автоматически применять преобразователи значений к определенным типам и именам свойств, чтобы облегчить себе жизнь.

7.9 Различные способы настройки первичного ключа

Вы уже видели подход «По соглашению» к настройке первичного ключа сущности. В этом разделе рассматривается обычная настройка первичного ключа – единственного ключа, для которого свойство .NET определяет имя и тип. Вам необходимо явно настроить первичный ключ в двух случаях:

- когда имя ключа не соответствует правилам именования «По соглашению»;
- когда первичный ключ состоит из нескольких свойств или столбцов. Такой ключ называется *составным ключом*.

Таблица со связью «многие ко многим» – пример того, где данный подход не работает. Для определения первичных ключей можно использовать два альтернативных подхода.

ПРИМЕЧАНИЕ Глава 8 посвящена настройке внешних ключей, поскольку они определяют связи, даже если они скалярного типа.

7.9.1 Настройка первичного ключа с помощью Data Annotations

Атрибут `[Key]` позволяет назначить одно свойство в качестве первичного ключа в классе. Используйте эту аннотацию, если вы не используете имя первичного ключа «По соглашению», как показано в следующем листинге. Этот простой код четко обозначает первичный ключ.

Листинг 7.7 Определение свойства в качестве первичного ключа с помощью аннотации `[Key]`

```
private class SomeEntity
{
    [Key]
    public int NonStandardKeyName { get; set; }

    public string MyString { get; set; }
}
```

Атрибут [Key] сообщает EF Core, что данное свойство – это первичный ключ

Обратите внимание, что атрибут [Key] нельзя использовать для составных ключей. В более ранних версиях EF Core можно было определять составные ключи с помощью атрибутов [Key] и [Column], но эта функция была удалена.

7.9.2 Настройка первичного ключа через Fluent API

Первичный ключ также можно настроить через Fluent API, что полезно для первичных ключей, не соответствующих шаблонам «По соглашению». В следующем листинге показаны два первичных ключа, настраиваемых методом Fluent API HasKey. Первый ключ – это отдельный первичный ключ с нестандартным именем в классе сущности SomeEntity, а второй – составной первичный ключ, состоящий из двух столбцов, в связующей таблице BookAuthor.

Листинг 7.8 Использование Fluent API для настройки первичных ключей для двух классов сущностей

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<SomeEntity>()
        .HasKey(x => x.NonStandardKeyName);

    modelBuilder.Entity<BookAuthor>()
        .HasKey(x => new {x.BookId, x.AuthorId});

    //… Остальные параметры конфигурации удалены;
}
```

Определяет обычный первичный ключ, состоящий из одного столбца. Используйте HasKey, если имя вашего ключа не соответствует установленным по умолчанию значениям «По соглашению»

Использует анонимный объект для определения двух (и более) свойств для формирования составного ключа. Порядок, в котором свойства указаны в анонимном объекте, определяет их порядок в составном ключе

Для составных ключей не существует версии «По соглашению», поэтому нужно использовать метод Fluent API HasKey.

7.9.3 Настройка сущности как класса с доступом только на чтение

В некоторых сложных ситуациях у класса сущности может не быть первичного ключа. Вот три примера:

- *вам нужно определить класс сущности с доступом только на чтение*. Если у класса сущности нет первичного ключа, то EF Core будет рассматривать его как класс с доступом только на чтение;
- *вам нужно отобразить класс сущности в представление SQL с доступом только на чтение*. *Представления SQL* – это запросы, ко-

торые работают как таблицы SQL. Смотрите эту статью для получения дополнительной информации: http://mng.bz/6g6y;
- *вам нужно отобразить класс сущности в запрос SQL с помощью команды Fluent API* `ToSqlQuery`. Метод `ToSqlQuery` позволяет определить запрос SQL, который будет выполняться во время чтения этого класса сущности.

Чтобы явно установить класс сущности как класс с доступом только на чтение, можно использовать команду Fluent API `HasNoKey()` или применить к классу атрибут `[Keyless]`. А если у класса сущности нет первичного ключа, нужно пометить его как класс с доступом только на чтение, используя любой из двух подходов. Любая попытка изменить базу данных с помощью класса сущности без первичного ключа завершится исключением. EF Core делает это, потому что не может выполнить обновление без ключа – это один из способов определения класса сущности как класса с доступом только на чтение. Другой способ – отобразить сущность в представление SQL с помощью метода Fluent API `ToView("ViewNameString")`, как показано ниже:

```
modelBuilder.Entity<MyEntityClass>()
    .ToView("MyView");
```

EF Core выдаст исключение, если вы попытаетесь изменить базу данных через класс сущности, который отображен в представление. Если вы хотите отобразить класс сущности в обновляемое представление – представление SQL, которое можно обновить, – следует использовать команду `ToTable`.

7.10 Добавление индексов в столбцы базы данных

В реляционных базах данных есть инструмент под названием *индекс*, обеспечивающий более быстрый поиск и сортировку строк по столбцу или столбцам, добавленным в него. Кроме того, на индекс могут накладываться ограничения, гарантирующие уникальность каждой записи в нем. Так, например, первичный ключ получает уникальный индекс, чтобы гарантировать, что для каждой строки в таблице он разный.

Индекс в столбец можно добавить с помощью Fluent API и атрибутов, как показано в табл. 7.2. Он ускорит быстрый поиск и сортировку, а если добавить уникальное ограничение, то база данных гарантирует, что значение столбца в каждой строке будет разным.

СОВЕТ Не забывайте, что можно связывать методы Fluent API в цепочку, чтобы смешивать и сочетать эти методы.

Таблица 7.2 Добавление индекса в столбец

Действие	Fluent API
Добавить индекс, Fluent	`modelBuilder.Entity<MyClass>()` `  .HasIndex(p => p.MyProp);`
Добавить индекс, атрибут	`[Index(nameof(MyProp))]` `public class MyClass …`
Добавить индекс, несколько столбцов	`modelBuilder.Entity<Person>()` `  .HasIndex(p => new {p.First, p.Surname});`
Добавить индекс, несколько столбцов, атрибут	`[Index(nameof(First), nameof(Surname)]` `public class MyClass …`

Добавить уникальный индекс, Fluent	`modelBuilder.Entity<MyClass>()`
	`  .HasIndex(p => p.BookISBN)`
	`  .IsUnique();`

Добавить уникальный индекс, атрибут	`[Index(nameof(MyProp), IsUnique = true)]` `public class MyClass …`

Добавить именованный индекс, Fluent	`modelBuilder.Entity<MyClass>()`
	`  .HasIndex(p => p.MyProp)`
	`  .HasDatabaseName("Index_MyProp");`

Некоторые базы данных позволяют указать отфильтрованный или частичный индекс для игнорирования определенных ситуаций с помощью оператора `WHERE`. Например, можно установить уникальный отфильтрованный индекс, который будет игнорировать любые мягко удаленные элементы. Чтобы настроить отфильтрованный индекс, используется метод Fluent API `HasFilter`, содержащий выражение SQL, которое определяет, должен ли индекс обновляться значением. В следующем фрагменте кода приведен пример, обеспечивающий, что свойство `MyProp` будет содержать уникальное значение, если только столбец таблицы `SoftDeleted` не имеет значения `true`:

```
modelBuilder.Entity<MyClass>()
    .HasIndex(p => p.MyProp)
    .IsUnique()
    .HasFilter("NOT SoftDeleted");
```

ПРИМЕЧАНИЕ При использовании поставщика SQL Server EF добавляет фильтр `IS NOT NULL` для всех столбцов, допускающих значение NULL, которые являются частью уникального индекса. Можно переопределить это соглашение, предоставив `null` параметру `HasFilter`: `HasFilter(null)`.

7.11 Настройка именования на стороне базы данных

Если вы создаете новую базу данных, то можно использовать имена по умолчанию для различных частей базы данных. Но если у вас есть уже существующая база данных или база должна быть доступна для существующей системы, которую нельзя изменить, то, скорее всего,

вам потребуется использовать определенные имена для *схемы*, таблиц и столбцов базы данных.

> **ОПРЕДЕЛЕНИЕ** *Схема* – это то, как организовано хранение данных внутри базы данных: в виде таблиц, столбцов, ограничений и т. д. В некоторых базах данных, таких как SQL Server, она также применяется для предоставления пространства имен определенной группе данных, которую разработчик базы данных использует для разбиения базы на логические группы.

7.11.1 Настройка имен таблиц

По соглашению имя таблицы задается именем свойства `DbSet<T>` в DbContext, или, если это свойство не определено, таблица использует имя класса. Например, в DbContext нашего приложения Book App мы определили свойство Books, `DbSet<Book>`, поэтому в качестве имени таблицы базы данных используется Books. И наоборот, мы не определили свойство `DbSet<T>` для класса сущности `Review` в DbContext, поэтому в качестве имени таблицы использовалось имя класса – Review.

Если в вашей базе данных есть определенные имена таблиц, которые не соответствуют правилам именования по соглашению, – например, если имя таблицы нельзя преобразовать в допустимое имя переменной .NET, потому что в нем есть пробел, – то можно использовать Data Annotations или Fluent API, чтобы задать имя таблицы. В табл. 7.3 показаны два подхода, чтобы задать имя таблицы.

Таблица 7.3 Два способа явно настроить имя таблицы для класса сущности

Метод конфигурации	Пример: установка имени «XXX» для таблицы класса Book
Data Annotations	`[Table("XXX")]` `public class Book … etc.`
Fluent API	`modelBuilder.Entity<Book>().ToTable("XXX");`

7.11.2 Настройка имени схемы и группировки схем

Некоторые базы данных, такие как SQL Server, позволяют группировать таблицы, используя имя схемы. У вас может быть две таблицы с одинаковым именем, но разными именами схемы: например, таблица Books с именем схемы Display отличается от таблицы Books с именем схемы Order.

По соглашению имя схемы устанавливается поставщиком базы данных, потому что некоторые базы данных, такие как SQLite и MySQL, не поддерживают схемы. В случае с SQL Server, которая поддерживает схемы, имя схемы по умолчанию – *dbo*. Имя схемы по умолчанию можно изменить только через Fluent API, используя следующий фрагмент кода в методе `OnModelCreating` DbContext приложения:

```
modelBuilder.HasDefaultSchema("NewSchemaName");
```

В табл. 7.4 показано, как задать имя схемы для таблицы. Этот подход используется, если ваша база данных разбита на логические группы, например продажи, производство, счета и т. д., и таблица должна быть назначена схеме.

Таблица 7.4. Установка имени схемы для конкретной таблицы

Метод конфигурации	Пример: установка названия схемы sales для таблицы
Data Annotations	`[Table("SpecialOrder", Schema = "sales")]` `class MyClass … etc.`
Fluent API	`modelBuilder.Entity<MyClass>()` `.ToTable("SpecialOrder", schema: "sales");`

7.11.3 Настройка имен столбцов базы данных в таблице

По соглашению имя столбца в таблице совпадает с именем свойства. Если в вашей базе данных есть имя, которое нельзя представить как допустимое имя переменной .NET или которое не подходит для использования в приложении, то можно задать имена столбцов с помощью Data Annotations или Fluent API. В табл. 7.5 показаны оба подхода.

Таблица 7.5 Два способа настройки имени столбца

Метод конфигурации	Задаем для столбца свойства `BookId` имя `SpecialCol`
Data Annotations	`[Column("SpecialCol")]` `public int BookId { get; set; }`

Fluent API	`modelBuilder.Entity<MyClass>()`
	`.Property(b => b.BookId)`
	`.HasColumnName("SpecialCol");`

7.12 Настройка глобальных фильтров запросов

У многих приложений, например приложений ASP.NET Core, есть инструменты безопасности, которые контролируют, к каким представлениям и элементам управления пользователь может получить доступ. В EF Core есть похожая функция – *глобальные фильтры запросов* (сокращенно – *фильтры запросов*). Их можно использовать для создания мультитенантного приложения. Этот тип приложения хранит данные для разных пользователей в одной базе данных, но каждый пользователь может видеть только те данные, к которым ему разрешен доступ. Еще один вариант использования – реализация функции мягкого удаления; вместо удаления данных из базы данных можно использовать фильтр запросов, чтобы мягко удаленная строка исчезла, а данные остались, если вам потребуется восстановить их позже.

Я обнаружил, что фильтры запросов полезны во многих клиентских задачах, поэтому включил подробный раздел под названием «Использование глобальных фильтров запросов в реальных ситуа-

циях» в главу 6 (раздел 6.1.6). В нем содержится информация о том, как настроить фильтры запросов, поэтому, пожалуйста, поищите там эту информацию. В разделе 7.16.4 я покажу, как автоматизировать их настройку. Это гарантирует, что вы не забудете добавить важный фильтр к одному из классов сущностей.

7.13 Применение методов Fluent API в зависимости от типа поставщика базы данных

Поставщики баз данных EF Core предоставляют способ определить, какой поставщик базы данных используется при создании экземпляра DbContext приложения. Этот подход полезен для таких ситуаций, как использование, скажем, базы данных SQLite для модульных тестов, когда рабочая база данных находится на SQL Server и нужно что-то изменить, чтобы заставить свои модульные тесты работать.

К примеру, SQLite не полностью поддерживает некоторые типы .NET, например `decimal`, поэтому если вы попытаетесь отсортировать `decimal` свойство в базе данных SQLite, то получите исключение, в котором говорится, что вы не получите корректный результат из базы данных. Один из способов обойти эту проблему – преобразовать тип `decimal` в `double` при использовании SQLite; он не будет точным, но может сгодиться для контролируемого набора модульных тестов.

Каждый поставщик базы данных предоставляет метод расширения, который возвращает значение `true`, если база данных соответствует этому поставщику. Например, у поставщика базы данных SQL Server есть метод `IsSqlServer()`; у поставщика базы данных SQLite – метод `IsSqlite()` и т. д. Еще один подход – использовать свойство `ActiveProvider` в классе `ModelBuilder`, возвращающее строку, которая является именем пакета NuGet поставщика базы данных, например `Microsoft.EntityFrameworkCore.SqlServer`.

Следующий листинг представляет собой пример замены типа свойств с decimal на double, если база данных – это SQLite. Этот код позволяет использовать в запросе метод приложения Book App `OrderBooksBy` для базы данных SQLite в памяти.

Листинг 7.9 Использование команд поставщика базы данных для установки типа столбца

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
     //… Помещаем сюда свою обычную конфигурацию;
     if (Database.IsSqlite())
     {
```

IsSqlite вернет true, если база данных, указанная в параметрах, – это SQLite

```
            modelBuilder.Entity<Book>()
                .Property(e => e.Price)
                .HasConversion<double>();
            modelBuilder.Entity<PriceOffer>()
                .Property(e => e.NewPrice)
                .HasConversion<double>();
    }
}
```

Вы изменяете типы двух свойств с decimal на double, чтобы модульный тест, выполняющий сортировку по этим значениям, не вызвал исключение

В EF Core 5 добавлен метод `IsRelational()`, возвращающий значение `false` для поставщиков баз данных, которые не являются реляционными, например Cosmos Db. В документации EF Core можно найти несколько методов Fluent API, специфичных для конкретной базы данных, например метод `IsMemoryOptimized` поставщика SQL Server.

> **ПРИМЕЧАНИЕ** Хотя и можно использовать этот подход для создания миграций для разных типов баз данных, делать этого не рекомендуется. Команда EF Core предполагает, что вы будете создавать отдельные миграции для каждого типа базы данных и сохранять их в отдельные каталоги. Для получения дополнительной информации см. главу 9.

7.14 Теневые свойства: сокрытие данных столбца внутри EF Core

> **EF6** В EF6.x существовало понятие теневых свойств, но они использовались только внутри него для обработки отсутствующих внешних ключей. В EF Core теневые свойства стали полноценной функцией, которую вы можете использовать.

Теневые свойства позволяют получить доступ к столбцам базы данных, не добавляя их в класс сущности как свойства. Они дают возможность «скрыть» данные, которые не считаются частью обычного использования класса сущности. Все дело в лучших практиках разработки программного обеспечения: вы позволяете объектам верхних уровней получать доступ только к тем данным, которые им нужны, и скрываете все, о чем они не должны знать. Приведу два примера, которые показывают, когда можно использовать теневые свойства:

- обычно необходимо отслеживать, кем и когда были изменены данные, возможно в целях аудита или для понимания поведения клиентов. Данные отслеживания, которые вы получите, не связаны с основным использованием класса, поэтому, возможно, вы решите реализовать их с помощью теневых свойств, доступ к которым можно получить за пределами класса сущности;
- когда вы настраиваете связи между классами сущностей, у которых еще не заполнены свойства внешнего ключа, EF Core должен

добавить эти свойства, чтобы связь работала, и он делает это, используя теневые свойства. Этой теме посвящена глава 8.

7.14.1 Настройка теневых свойств

Существует подход «По соглашению» к настройке теневых свойств, но поскольку он относится только к связям, я объясняю его в главе 8. Еще один метод – использовать Fluent API. Можно добавить новое свойство, используя метод Fluent API `Property<T>`. Поскольку вы настраиваете теневое свойство, в самом классе сущности не будет свойства с таким именем, поэтому нужно использовать метод `Property<T>`, который принимает тип .NET и имя теневого свойства. В следующем листинге показана настройка теневого свойства `UpdatedOn` типа `DateTime`.

Листинг 7.10 Создание теневого свойства `UpdatedOn` с помощью Fluent API

```
public class Chapter06DbContext : DbContext
{
    …

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<MyEntityClass>()
            .Property<DateTime>("UpdatedOn");
    …
    }
}
```

Использует метод Property<T> для установки типа теневого свойства

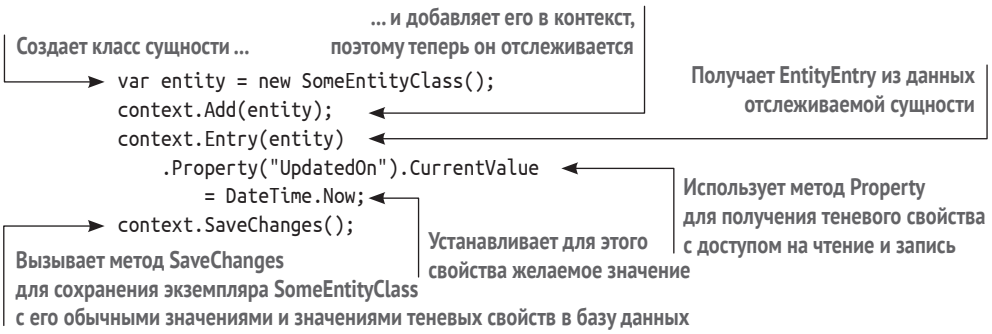
По соглашению имя столбца таблицы, в которое отображается теневое свойство, совпадает с именем теневого свойства. Можно переопределить этот параметр, добавив метод `HasColumnName` после метода `Property`.

ПРЕДУПРЕЖДЕНИЕ Если свойство с таким именем уже существует в классе сущности, то конфигурация будет использовать это свойство вместо создания теневого свойства.

7.14.2 Доступ к теневым свойствам

Поскольку теневые свойства не отображаются в свойство класса, доступ к ним можно получить только напрямую через EF Core. Для этого нужно использовать команду EF Core `Entry(myEntity).Property("MyPropertyName").CurrentValue`. `CurrentValue` является свойством с доступом на чтение и запись, как показано в следующем листинге.

Листинг 7.11 Использование `Entry(inst).Property(name)` для настройки теневого свойства



Если вы хотите получить значение теневого свойства в загруженной сущности, используйте команду `context.Entry(entityInstance).Property("PropertyName").CurrentValue`. Но нужно читать сущность как отслеживаемую, без использования метода `AsNoTracking` в запросе. Метод `Entry(<entityInstance>).Property` использует данные отслеживаемой сущности внутри EF Core для хранения значения, поскольку оно не хранится в экземпляре класса сущности.

В запросах LINQ для доступа к теневому свойству используется другой метод: команда `EF.Property`. Можно выполнить сортировку по теневому свойству `UpdatedOn`, например используя следующий фрагмент запроса, метод `EF.Property` выделен полужирным шрифтом:

```
context.MyEntities
    .OrderBy(b => EF.Property<DateTime>(b, "UpdatedOn"))
    .ToList();
```

7.15 Резервные поля: управление доступом к данным в классе сущности

> **EF6** Резервные поля недоступны в EF6. Эта функция EF Core обеспечивает больший уровень контроля над доступом к данным, которого не было у пользователей EF6.x.

Как было показано ранее, столбцы в таблице базы данных обычно отображаются в свойство класса сущности с методами чтения и записи – `public int MyProp { get; set; }`. Но вы также можете отобразить закрытое поле в свою базу данных. Эта функция называется *резервным полем*. Она дает вам больше контроля над тем, как данные базы данных считываются или редактируются программным обеспечением.

Подобно теневым свойствам, резервные поля скрывают данные, но делают это по-другому. В случае с теневыми свойствами данные скрыты внутри данных EF Core, а резервные поля скрывают данные

внутри класса сущности, поэтому классу сущности проще получить доступ к резервному полю внутри класса. Вот несколько примеров, в которых можно использовать эти поля:

- *сокрытие конфиденциальных данных* – сокрытие даты рождения человека в закрытом поле и обеспечение доступности его возраста в годах для остальной части программного обеспечения;
- *перехват изменений* – обнаружение изменения значения свойства путем сохранения данных в закрытом поле и добавления кода в метод записи для обнаружения изменений. Мы будем использовать этот прием в главе 12, когда будем использовать изменение свойства для запуска события;
- *создание классов сущностей предметно-ориентированного проектирования* (*DDD*) – создание DDD классов сущностей, в которых свойства всех классов сущностей должны быть свойствами с доступом только на чтение. Резервные поля позволяют заблокировать навигационные свойства коллекции, как описано в разделе 8.7.

Но, прежде чем перейти к сложным версиям, начнем с простейшей формы резервных полей, где методы чтения и записи свойства обращаются к полю.

7.15.1 Создание простого резервного поля, доступного через свойство чтения/записи

В следующем фрагменте кода показано строковое свойство `MyProperty`, в котором строковые данные хранятся в закрытом поле. Эта форма резервного поля ничем особо не отличается от использования обычного свойства, но в этом примере показана концепция свойства, связанного с закрытым полем:

```
public class MyClass
{
    private string _myProperty;
    public string MyProperty
    {
        get { return _myProperty; }
        set { _myProperty = value; }
    }
}
```

Конфигурация EF Core «По соглашению» найдет тип резервного поля и сконфигурирует его как резервное поле (см. раздел 7.15.4, где описаны опции конфигурации резервного поля), и по умолчанию EF Core будет читать или записывать данные базы данных в это закрытое поле.

7.15.2 Создание столбца с доступом только на чтение

Создание столбца с доступом только на чтение – наиболее очевидный вариант использования, хотя его также можно реализовать с по-

мощью свойства с закрытым доступом на запись (см. раздел 7.3.2). Если у вас есть столбец в базе данных, который вам нужно прочитать, но вы не хотите, чтобы программа могла изменить его, резервное поле – отличное решение. В этом случае можно создать закрытое поле и использовать открытое свойство только с методом чтения для получения значения. В следующем фрагменте кода приводится пример:

```
public class MyClass
{
    private string _readOnlyCol;
    public string ReadOnlyCol => _readOnlyCol;
}
```

Что-то должно установить свойство столбца, например установить значение по умолчанию в столбце базы данных (рассматривается в главе 9) или с помощью какого-то внутреннего метода базы данных.

7.15.3 Сокрытие даты рождения внутри класса

Сокрытие даты рождения – возможный вариант использования резервных полей. В этом случае, исходя из соображений безопасности, дату рождения можно установить, но из класса сущности можно получить только возраст. В следующем листинге показано, как сделать это в классе `Person`, используя закрытое поле `_dateOfBirth`, а затем предоставляя метод для его установки и свойство для вычисления возраста.

Листинг 7.12 Использование резервного поля для скрытия конфиденциальных данных от обычного доступа

```
public class Person
{
    private DateTime _dateOfBirth;

    public void SetDateOfBirth(DateTime dateOfBirth)
    {
        _dateOfBirth = dateOfBirth;
    }

    public int AgeYears =>
        Years(_dateOfBirth, DateTime.Today);

    //Спасибо пользователю dana из stackoverflow
    //См. http://stackoverflow.com/a/4127477/1434764
    private static int Years(DateTime start, DateTime end)
    {
        return (end.Year - start.Year - 1) +
               (((end.Month > start.Month) ||
                 ((end.Month == start.Month)
                  && (end.Day >= start.Day)))
                  ? 1 : 0);
    }}
```

Закрытое резервное поле, к которому нельзя получить доступ напрямую в других классах .NET

Позволяет установить резервное поле

Вы можете узнать возраст человека, но не его точную дату рождения

ПРИМЕЧАНИЕ В предыдущем примере нужно использовать Fluent API для создания переменной только для резервного поля (рассматривается в разделе 7.15.2), потому что EF Core не может найти это поле, используя подход «По соглашению».

С точки зрения класса поле `_dateOfBirth` скрыто, но вы все равно можете получить доступ к столбцу таблицы с помощью различных методов EF Core точно так же, как вы получали доступ к теневым свойствам: с помощью метода `EF.Property<DateTime>(entity, "_dateOfBirth")`.

Резервное поле `_dateOfBirth` не полностью защищено от разработчика, но это и не является целью. Идея состоит в том, чтобы удалить данные о дате рождения из обычных свойств, чтобы они не были доступны для клиентов класса.

7.15.4 Настройка резервных полей

Увидев резервные поля в действии, можно настроить их по соглашению, через Fluent API, а теперь и в EF Core 5, используя Data Annotations. Подход «По соглашению» работает хорошо, но он полагается на то, что у класса есть свойство, соответствующее полю по типу и соглашению об именах. Если поле не соответствует имени или типу свойства или у него нет соответствующего свойства, как, например, в случае с `_dateOfBirth`, то необходимо настроить резервные поля с помощью Data Annotations или Fluent API. В следующих разделах описаны различные подходы к настройке.

НАСТРОЙКА РЕЗЕРВНЫХ ПОЛЕЙ ПО СОГЛАШЕНИЮ

Если ваше резервное поле связано с допустимым свойством (см. раздел 7.3.2), это поле можно настроить по соглашению. Правила конфигурации «По соглашению» гласят, что у закрытого поля должно быть одно из следующих имен, которые соответствуют свойству в том же классе:

- `_<property name>` (например, `_MyProperty`);
- `_<camel-cased property name>` (например, `_myProperty`);
- `m_<property name>` (например, `m_MyProperty`);
- `m_<camel-cased property name>` (например, `m_myProperty`).

ОПРЕДЕЛЕНИЕ *Camel case* – это соглашение, при котором имя переменной начинается со строчной буквы, а прописная буква используется для начала каждого последующего слова в имени, например `thisIsCamelCase`.

НАСТРОЙКА РЕЗЕРВНЫХ ПОЛЕЙ С ПОМОЩЬЮ DATA ANNOTATIONS

Новинка EF Core 5 – атрибут `BackingField`, позволяющий связать свойство с закрытым полем в классе сущности. Это полезный атри-

бут, если вы не используете стиль именования резервных полей «По соглашению», как в этом примере:

```
private string _fieldName;
[BackingField(nameof(_fieldName))]
public string PropertyName
{
    get { return _fieldName; }
}

public void SetPropertyNameValue(string someString)
{
    _fieldName = someString;
}
```

НАСТРОЙКА РЕЗЕРВНЫХ ПОЛЕЙ С ПОМОЩЬЮ FLUENT API

Есть несколько способов настройки резервных полей через Fluent API. Начнем с самого простого и перейдем к более сложному. В каждом примере показан метод `OnModelCreating` внутри DbContext, при этом настраивается только часть, касающаяся поля:

- *установка имени резервного поля* – если имя резервного поля не следует соглашениям EF Core, то необходимо указать имя поля, используя Fluent API. Например:

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(b => b.MyProperty)
        .HasField("_differentName");
    …
    }
```

- *предоставление только имени поля* – в этом случае если у вас есть свойство с корректным именем, то по соглашению EF Core будет ссылаться на свойство, а имя свойства будет использоваться для столбца базы данных. Вот пример:

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property("_dateOfBirth")
        .HasColumnName("DateOfBirth");
    …
}
```

Если методы чтения и записи свойства не найдены, то поле все равно будет отображено в столбец с использованием его имени, в нашем примере это `_dateOfBirth`, но, скорее всего, это не то имя, которое нужно нам для столбца. Поэтому мы добавляем метод Fluent API

`HasColumnName` для получения наиболее подходящего имени. Обратная сторона состоит в том, что нам по-прежнему необходимо ссылаться на данные в запросе по имени поля (в данном случае `_dateOfBirth`), что не слишком удобно и не очевидно.

ДОПОЛНИТЕЛЬНО: НАСТРОЙКА ЧТЕНИЯ И ЗАПИСИ ДАННЫХ В РЕЗЕРВНОМ ПОЛЕ

Начиная с версии EF Core 3 режим доступа к базе данных по умолчанию для полей резервного копирования – чтение и запись в поле. Этот режим работает почти во всех случаях, но если вы хотите изменить его, то можете сделать это, используя метод Fluent API `UsePropertyAccessMode`. В следующем фрагменте кода мы предписываем EF Core попробовать использовать свойство для чтения и записи, но если в свойстве отсутствует метод записи, то EF Core заполнит поле при чтении из базы данных:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(b => b.MyProperty)
        .HasField("_differentName")
        .UsePropertyAccessMode(PropertyAccessMode.PreferProperty);
    …
}
```

> **СОВЕТ** Чтобы увидеть различные режимы доступа для резервного поля, воспользуйтесь функцией Visual Studio *intellisense* для просмотра комментариев к каждому значению перечисления `PropertyAccessMode`.

7.16 Рекомендации по использованию конфигурации EF Core

Есть очень много способов настроить EF Core, некоторые из которых дублируют друг друга, поэтому не всегда очевидно, какой из трех подходов следует использовать для каждой части конфигурации. Вот предлагаемые подходы:

- по возможности начните с использования подхода «По соглашению», потому что это быстро и просто;
- используйте атрибуты валидации – `MaxLength`, `Required` и т. д. – из подхода Data Annotations, поскольку они полезны при валидации данных;
- для всего остального используйте Fluent API, потому что у него имеется наиболее полный набор команд. Но подумайте о написании кода для автоматизации распространенных настроек, на-

пример применение «исправления UTC» ко всем свойствам `DateTime`, чье имя заканчивается на `"UTC"`.

В следующих разделах представлены более подробные объяснения моих рекомендаций по настройке EF Core.

7.16.1 Сначала используйте конфигурацию «По соглашению»

EF Core выполняет приличный объем работы по настройке большинства стандартных свойств, поэтому всегда начинайте с этого подхода. В первой части книги мы создали всю исходную базу данных, используя подход «По соглашению», за исключением составного ключа из класса сущности `BookAuthor`.

Это быстрый и простой подход. В главе 8 вы увидите, что большинство связей можно настроить исключительно с помощью правил именования «По соглашению», что сэкономит много времени. Изучение возможностей этого подхода значительно сократит объем кода, который вам нужно будет написать.

7.16.2 По возможности используйте Data Annotations

Хотя вы и можете делать такие вещи, как ограничивать размер строкового свойства как с помощью Data Annotations, так и Fluent API, я рекомендую использовать первый способ по следующим причинам:

- *их можно использовать при проверке клиентской части*. Хотя EF Core не проверяет класс сущности перед сохранением его в базе данных, другие части системы могут использовать Data Annotations для валидации. Например, ASP.NET Core использует Data Annotations для проверки вводимых пользователем данных, поэтому если вы передаете такие данные непосредственно в класс сущности, то атрибуты валидации будут полезны. Или если вы используете отдельные классы ASP.NET ViewModel либо DTO, то можете скопировать и вставить свойства с их атрибутами проверки;
- *можно добавить проверку в метод EF Core* `SaveChanges`. Использование валидации данных для вывода проверок из бизнес-логики может ее упростить. В главе 4 показано, как добавить проверку классов сущностей при вызове метода `SaveChanges`;
- *Data Annotations представляют собой отличные комментарии*. Атрибуты, включающие в себя Data Annotations, – это константы времени компиляции; их легко увидеть и понять.

7.16.3 Используйте Fluent API для всего остального

Обычно я использую Fluent API для настройки отображения столбцов базы данных (имя, тип данных и т. д.), когда есть отличия от значений по умолчанию. Для этого можно было бы использовать и подход

с Data Annotations, но я стараюсь скрывать такие вещи внутри метода `OnModelCreating`, потому что это проблемы, связанные с реализацией базы данных, а не проблемы, относящиеся к структуре приложения. Хотя такая практика – скорее рекомендация, нежели правило, так что решать вам. В разделе 7.16.4 описывается, как автоматизировать свои конфигурации Fluent API, что сэкономит вам время, а также гарантирует, что все ваши правила конфигурации будут применены к каждому подходящему классу или свойству.

7.16.4 Автоматизируйте добавление команд Fluent API по сигнатурам класса или свойства

У команд Fluent API есть одна полезная функция, позволяющая писать код для поиска и настройки определенных конфигураций на основе типа класса или свойства, имени и т. д. В реальном приложении могут быть сотни свойств `DateTime`, которым требуется исправление UTC, использованное нами в листинге 7.6. Вместо того чтобы добавлять конфигурацию для каждого свойства вручную, не лучше ли найти каждое свойство, для которого требуется исправление UTC, и применить его автоматически? Именно это мы и сделаем.

Автоматизация поиска и добавления конфигураций основана на типе `IMutableModel`, доступ к которому можно получить в методе `OnModelCreating`. Этот тип дает доступ ко всем классам, отображаемым EF Core в базу данных, и каждый `IMutableEntityType` позволяет получить доступ к свойствам. Большинство параметров конфигурации можно применить с помощью методов в этих двух интерфейсах, но некоторые, например фильтры запросов, требуют дополнительной работы.

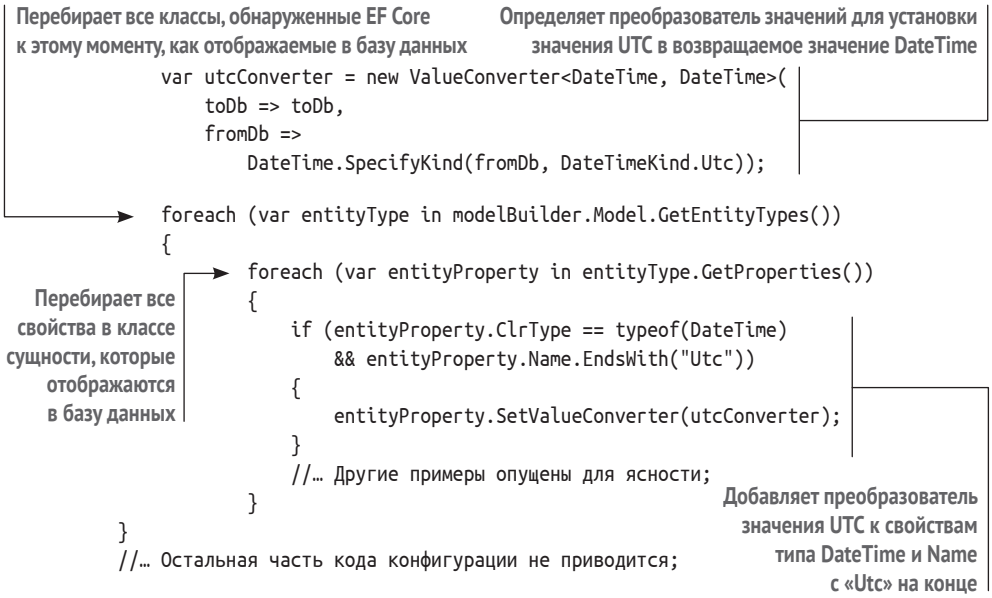
Для начала мы напишем код, который будет перебирать классы сущностей и их свойства и добавлять одну конфигурацию, как показано в листинге 7.13. Этот подход определяет способ автоматизации конфигураций, а в последующих примерах мы добавим дополнительные команды для выполнения дополнительных конфигураций.

В следующем примере к `DateTime` добавляется преобразователь значений, который применяет исправление UTC, показанное в листинге 7.6. Но в следующем листинге преобразователь значения применяется к каждому свойству, которое представляет собой `DateTime` с именем, оканчивающимся на `"Utc"`.

Листинг 7.13 Применение преобразователя значений к любому свойству `DateTime`, оканчивающемуся на `"Utc"`

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
```

Команды Fluent API применяются в методе OnModelCreating



В листинге 7.13 показана настройка только одного свойства, но обычно присутствует множество настроек Fluent API. В этом примере мы сделаем следующее:

1. Добавим преобразователь значения UTC в свойства типа `DateTime`, у которых `Name` оканчивается на `"UTC"`.
2. Установим десятичную точность / масштаб, если в имени свойства содержится `"Price"`.
3. Сделаем так, чтобы любые строковые свойства, у которых имя оканчивается на `"URL"`, были сохранены как ASCII, т. е. `varchar(nnn)`.

В следующем фрагменте кода показан код внутри метода `OnModelCreating` в DbContext приложения Book App, чтобы добавить эти три параметра конфигурации:

```
foreach (var entityType in modelBuilder.Model.GetEntityTypes())
{
    foreach (var entityProperty in entityType.GetProperties())
    {
        if (entityProperty.ClrType == typeof(DateTime)
            && entityProperty.Name.EndsWith("Utc"))
               {
            entityProperty.SetValueConverter(utcConverter);
        }

        if (entityProperty.ClrType == typeof(decimal)
            && entityProperty.Name.Contains("Price"))
        {
            entityProperty.SetPrecision(9);
            entityProperty.SetScale(2);
        }
```

```
            if (entityProperty.ClrType == typeof(string)
                && entityProperty.Name.EndsWith("Url"))
            {
                entityProperty.SetIsUnicode(false);
            }
        }
    }
```

Однако для некоторых конфигураций Fluent API требуется код, зависящий от конкретного класса. Фильтрам запросов, например, нужен запрос, который обращается к классам сущностей. В этом случае необходимо добавить к классу сущности интерфейс, к которому вы хотите добавить фильтр запросов, и динамически создать правильный запрос.

В качестве примера мы напишем код, позволяющий автоматически добавлять фильтр запросов `SoftDelete`, описанный в разделе 3.5.1, и фильтр запросов `UserId` из раздела 6.1.7. Из этих двух фильтров `UserId` более сложен, потому что ему необходимо получить текущий `UserId`, который меняется для каждого экземпляра DbContext приложения Book App. Можно сделать это двумя способами, но мы решили предоставить текущий экземпляр DbContext к запросу. В следующем листинге показан класс расширения `SoftDeleteQueryExtensions` и перечисление `MyQueryFilterTypes`, которое он использует.

Листинг 7.14 Перечисление и класс, используемые для настройки фильтров запросов для каждого совместимого класса

Третье необязательное свойство содержит копию текущего экземпляра DbContext, поэтому UserId будет актуальным

Вызываем этот метод, чтобы настроить фильтр запроса

Второй параметр позволяет выбрать тип фильтра запроса, который нужно добавить

Определяет различные типы LINQ-запроса для помещения в фильтр запросов

```
public enum MyQueryFilterTypes { SoftDelete, UserId }

public static class SoftDeleteQueryExtensions
{
    public static void AddSoftDeleteQueryFilter(
        this IMutableEntityType entityData,
        MyQueryFilterTypes queryFilterType,
        IUserId userIdProvider = null)
    {
        var methodName = $"Get{queryFilterType}Filter";
        var methodToCall = typeof(SoftDeleteQueryExtensions)
            .GetMethod(methodName,
                BindingFlags.NonPublic | BindingFlags.Static)
            .MakeGenericMethod(entityData.ClrType);
        var filter = methodToCall
            .Invoke(null, new object[] { userIdProvider });
        entityData.SetQueryFilter((LambdaExpression)filter);
```

Статический класс расширения

Первый параметр поступает из EF Core и позволяет добавить фильтр запроса

Создает правильно типизированный метод для создания выражения Where, чтобы использовать его в фильтре запросов

Использует фильтр, возвращаемый методом созданного типа в методе SetQueryFilter

```
            if (queryFilterType == MyQueryFilterTypes.SoftDelete)
                entityData.AddIndex(entityData.FindProperty(
                    nameof(ISoftDelete.SoftDeleted)));
            if (queryFilterType == MyQueryFilterTypes.UserId)
                entityData.AddIndex(entityData.FindProperty(
                    nameof(IUserId.UserId)));
        }

        private static LambdaExpression GetUserIdFilter<TEntity>(
            IUserId userIdProvider)
            where TEntity : class, IUserId
        {
            Expression<Func<TEntity, bool>> filter =
                x => x.UserId == userIdProvider.UserId;
            return filter;
        }

        private static LambdaExpression GetSoftDeleteFilter<TEntity>(
            IUserId userIdProvider)
            where TEntity : class, ISoftDelete
        {
            Expression<Func<TEntity, bool>> filter =
                x => !x.SoftDeleted;
            return filter;
        }
    }
```

Добавляет индекс для свойства UserId для повышения производительности

Добавляет индекс для свойства SoftDeleted для повышения производительности

Создает запрос, который является истинным, только если userId совпадает с идентификатором пользователя в классе сущности

Создает запрос, который является истинным, только если свойство SoftDeleted имеет значение false

Поскольку каждый запрос сущности, у которой есть фильтр запросов, будет содержать фильтр для этого свойства, код автоматически добавляет индекс для каждого свойства, которое используется в фильтре запросов. Этот метод улучшает производительность сущности. Наконец, в следующем листинге показано, как использовать код, приведенный в листинге 7.14, в DbContext приложения Book App для автоматизации конфигурации фильтров запросов.

Листинг 7.15 Добавление кода в DbContext для автоматизации настройки фильтров запросов

```
public class EfCoreContext : DbContext, IUserId
{
    public Guid UserId { get; private set; }

    public EfCoreContext(DbContextOptions<EfCoreContext> options,
        IUserIdService userIdService = null)
        : base(options)
    {
        UserId = userIdService?.GetUserId()
                 ?? new ReplacementUserIdService().GetUserId();
```

Содержит UserId, который используется в фильтре запросов, использующем интерфейс IUserId

Добавление IUserId в DbContext означает, что можно передать DbContext фильтру запросов UserId

Настраивает UserId. Если userIdService имеет значение null или возвращает значение null для UserId, устанавливаем замену для UserId

```
    }

    //DbSet удалены для ясности

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        // Остальной код конфигурации удален для ясности;

        foreach (var entityType in modelBuilder.Model.GetEntityTypes()
        {
            // Остальной код свойств удален для ясности;

            if (typeof(ISoftDelete)
                .IsAssignableFrom(entityType.ClrType))
            {
                entityType.AddSoftDeleteQueryFilter(
                    MyQueryFilterTypes.SoftDelete);
            }
            if (typeof(IUserId)
                .IsAssignableFrom(entityType.ClrType))
            {
                entityType.AddSoftDeleteQueryFilter(
                    MyQueryFilterTypes.UserId, this);
            }
        }
    }
```

Код автоматизации помещается в метод OnModelCreating

Перебирает все классы, которые EF Core обнаружил на данный момент отображенными в базу данных

Если класс реализует интерфейс ISoftDelete, ему требуется фильтр запросов SoftDelete

Добавляет в этот класс фильтр запросов с запросом, подходящим для SoftDelete

Если класс реализует интерфейс IUserId, ему нужен фильтр запросов IUserId

Добавляет в этот класс фильтр запросов UserId. Передача «this» позволяет получить доступ к текущему UserId

Для приложения Book App вся эта автоматизация является излишней, но в более крупных приложениях она может сэкономить много времени; что еще более важно, это гарантирует, что все настроено правильно. В завершение этого раздела приведу несколько рекомендаций и ограничений, о которых нужно знать, если вы собираетесь использовать данный подход:

- если вы запустите автоматический код Fluent API перед настройками, написанными вручную, то ваши конфигурации, написанные вручную, переопределят все автоматические настройки Fluent API. Но имейте в виду, что если существует класс сущности, который зарегистрирован только с помощью написанного вручную Fluent API, автоматический код его не увидит;
- команды конфигурации должны применять одни и те же конфигурации каждый раз, поскольку EF Core настраивает DbContext приложения только один раз – при первом обращении, – а после этого работает с кеш-версией.

Резюме

- При первом создании DbContext приложения EF Core настраивается самостоятельно, используя комбинацию из трех подходов: «По соглашению», Data Annotations и Fluent API.

- Конвертеры значений позволяют преобразовывать тип/значение программного обеспечения при записи и чтении из базы данных.
- Две функции EF Core, теневые свойства и резервные поля, позволяют скрывать данные из более высоких уровней кода и/или управлять доступом к данным в классе сущности. Используйте подход «По соглашению», чтобы настроить как можно больше, т. к. он прост и ускоряет написание кода.
- Если подход «По соглашению» не соответствует вашим потребностям, Data Annotations и/или Fluent API могут предоставить дополнительные команды, чтобы настроить, как EF Core будет отображать классы сущностей в базу данных и как будет обрабатывать эти данные.
- Помимо написания кода конфигурации вручную, можно добавить код для автоматической настройки классов сущностей и/или свойств на основе сигнатуры класса/свойств.

Для читателей, знакомых с EF6:

- базовый процесс настройки EF Core на первый взгляд похож на то, как работает EF6, но здесь есть значительное количество измененных или новых команд;
- EF Core может использовать классы конфигурации для хранения команд Fluent API для заданного класса сущности. Команды Fluent API предоставляют функцию, аналогичную классу EF6.x `EntityTypeConfiguration<T>`, но EF Core использует интерфейс `IEntityTypeConfiguration<T>`;
- EF Core представил множество дополнительных возможностей, которых нет в EF6, например преобразователи значений, теневые свойства и резервные поля. Все это приятные дополнения для EF.

8 Конфигурирование связей

В этой главе рассматриваются следующие темы:

- настройка связей с использованием подхода «По соглашению»;
- настройка связей с использованием Data Annotations;
- настройка связей с использованием Fluent API;
- пять других способов отображения сущностей в таблицы базы данных.

В главе 7 описано, как настроить скалярные (нереляционные) свойства. В этой главе описывается, как настроить связи в базе данных. Я предполагаю, что вы прочитали по крайней мере первую часть главы 7, потому что при настройке связей используются те же три подхода для отображения связей: «По соглашению», Data Annotations и Fluent API.

В этой главе рассказывается, как EF Core находит и настраивает связи между классами сущности с рекомендациями и примерами, иллюстрирующими настройку каждого типа связи: «один к одному», «один ко многим» и «многие ко многим». Используя подход «По соглашению», можно быстро настроить множество связей, но вы также узнаете обо всех параметрах конфигурации с помощью Data Annotations и Fluent API, которые позволяют точно определить, какое поведение связей вам нужно. Кроме того, мы рассмотрим особенности, которые

позволят улучшить связи с помощью дополнительных ключей, и альтернативные подходы к отображению таблиц. В завершение изучим пять способов отображения классов в базу данных.

8.1 Определение терминов, относящихся к связям

В этой главе рассматриваются различные части связей между сущностями, и нужны четкие термины, чтобы точно знать, о какой части идет речь. Эти термины показаны на рис. 8.1. Здесь используются классы сущностей `Book` и `Review` из нашего приложения Book App. Я буду ориентироваться на этот рисунок, приводя более подробное описание, чтобы термины, используемые в данной главе, были вам понятны.

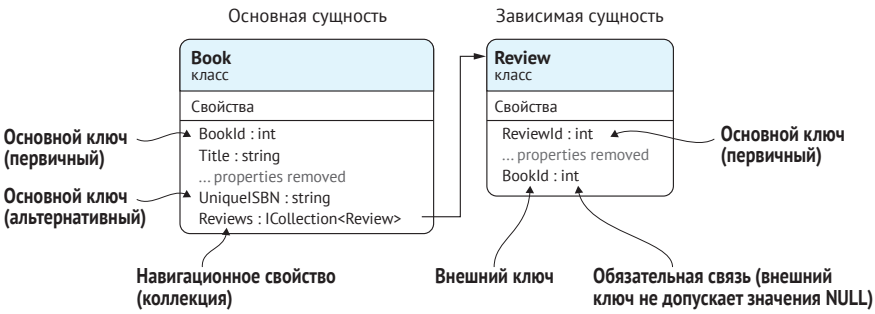


Рис. 8.1 Классы сущностей `Book` и `Review` показывают шесть терминов, используемых в этой главе для обсуждения связей: *основная сущность*, *зависимая сущность*, *основной ключ*, *навигационное свойство*, *внешний ключ* и *обязательная связь*. Не показана *необязательная связь*, описанная в разделе 2.1.1

Чтобы убедиться, что вам все понятно, приведу подробные описания:

- *основной ключ* – новый термин из документации EF Core, который относится к первичному ключу, определение которого приводится в первой части, либо к новому *альтернативному ключу*, у которого есть уникальное значение для каждой строки и который не является первичным ключом (см. раздел 8.8.3);

ПРИМЕЧАНИЕ На рис. 8.1 представлен пример альтернативного ключа `UniqueISBN`, который представляет собой уникальное значение для каждой сущности (ISBN – это международный стандартный книжный номер, уникальный для каждой книги).

- *основная сущность* – сущность, содержащая свойства основного ключа, на которые зависимая связь ссылается через внешний ключ (ключи) (рассматривается в главе 3);
- *зависимая сущность* – сущность, содержащая свойства внешнего ключа, которые ссылаются на основную сущность (рассматривается в главе 3);
- *основной ключ* – у сущности есть основной ключ, также известный как *первичный*, который уникален для каждой сущности, хранящейся в базе данных;
- *навигационное свойство* – термин из документации EF Core, обозначающий свойство, содержащее один класс сущности или коллекцию классов сущностей, которые EF Core использует для связывания классов сущностей;
- *внешний ключ* – его определение дано в разделе 2.1.3. Он содержит значение (значения) основного ключа строки базы данных, с которой он связан (или может быть null);
- *обязательная связь* – связь, в которой внешний ключ не допускает значения null (и должна присутствовать основная сущность);
- *необязательная связь* – связь, в которой внешний ключ допускает значение null (а основная сущность может отсутствовать).

ПРИМЕЧАНИЕ Основной и внешний ключи могут состоять из нескольких свойств или столбцов. Эти ключи называются *составными*. Вы уже видели один из таких ключей в разделе 3.4.4. У класса сущности `BookAuthor` есть составной первичный ключ, состоящий из `BookId` и `AuthorId`.

В разделе 8.4 вы увидите, что EF Core может находить и настраивать большинство связей по соглашению. В некоторых случаях ему требуется помощь, но обычно он может найти и настроить навигационные свойства за вас, если вы используете правила именования «По соглашению».

8.2 Какие навигационные свойства нам нужны?

При настройке связей между классами сущности следует руководствоваться бизнес-потребностями своего проекта. Можно добавить навигационные свойства на обоих концах связи, но это предполагает, что оба навигационных свойства одинаково полезны, хотя это не всегда так. Считается хорошей практикой предоставлять только те навигационные свойства, которые имеют смысл с точки зрения бизнеса или проектирования программного обеспечения.

В нашем приложении Book App, например, у класса сущности `Book` есть множество классов сущности `Review`, и каждый из них связан че-

рез внешний ключ с одной книгой. Следовательно, у вас может быть навигационное свойство типа `ICollection<Review>` в классе `Book` и навигационное свойство типа `Book` в классе `Review`. В этом случае у вас будет *полностью определенная связь*: связь с навигационными свойствами на обоих концах.

Но нужна ли она вам? С точки зрения проектирования программного обеспечения есть два вопроса, касающихся навигационных связей между `Book` и `Review`. Ответы на эти вопросы определяют, какие навигационные связи необходимо включить:

- должен ли класс сущности `Book` знать о классах сущности `Review`? Я отвечаю «да», потому что мы хотим рассчитывать средний балл по отзывам;
- должен ли класс сущности `Review` знать о классе сущности `Book`? Я отвечаю «нет», потому что в этом примере мы никак эти связи не используем.

Поэтому наше решение состоит в том, чтобы иметь только навигационное свойство `ICollection <Review>` в классе `Book`, как показано на рис. 8.1.

По моему опыту, навигационное свойство нужно добавлять только тогда, когда это имеет смысл с точки зрения бизнеса или когда вам нужно такое свойство для создания (с помощью метода `Add`) класса сущности со связью (см. раздел 6.2.1). Сведя к минимуму навигационные свойства, классы сущностей будет легче понять, и менее опытные разработчики не будут склонны использовать связи, которые не подходят для вашего проекта.

8.3 Настройка связей

Так же, как и в главе 7, в которой рассказывалось о настройке нереляционных свойств, в EF Core есть три способа настройки связей. Вот три подхода к настройке свойств, сфокусированных на связях:

- «*По соглашению*» – EF Core находит и настраивает связи путем поиска ссылок на классы, в которых есть первичный ключ;
- *Data Annotations* – эти аннотации можно использовать для обозначения внешних ключей и ссылок на связи;
- *Fluent API* – этот API предоставляет богатейший набор команд для полной настройки любых связей.

В следующих трех разделах подробно описывается каждый из этих подходов. Как вы увидите, подход «По соглашению» может автоматически настроить многие связи, если следовать его стандартам именования. С другой стороны, Fluent API позволяет определять каждую часть отношений вручную. Это может быть полезно, если у вас есть связь, которая выходит за рамки подхода «По соглашению».

8.4 Настройка связей по соглашению

Подход «По соглашению» реально экономит время, когда дело касается настройки связей. В EF6.x я кропотливо определял связи, потому что не полностью осознавал силу подхода «По соглашению». Теперь, когда я понимаю соглашения, то позволяю EF Core настраивать большинство связей самостоятельно, кроме тех немногих случаев, когда данный подход не работает. (Эти исключения перечислены в разделе 8.4.6.)

Правила просты, но требуется немного времени, чтобы усвоить принципы, согласно которым имя свойства, тип и допустимость значения null работают сообща. Надеюсь, что, прочитав этот раздел, вы сэкономите время при разработке следующего приложения, где используется EF Core.

8.4.1. Что делает класс классом сущности?

В главе 2 термин *класс сущности* определен как обычный класс .NET, который отображен EF Core в базу данных. Здесь нужно определить, как EF Core находит и идентифицирует класс как класс сущности с использованием подхода «По соглашению».

На рис. 7.1 показаны три способа настройки EF Core. Ниже приводится резюме этого процесса, который теперь сосредоточен на поиске связей и навигационных свойств:

1. EF Core сканирует DbContext в поисках любых открытых свойств `DbSet<T>`. Предполагается, что классы `T` в свойствах `DbSet<T>` – это классы сущностей.
2. EF Core также проверяет каждое открытое свойство в классах, найденных на этапе 1, и ищет свойства, которые могут быть навигационными. Свойства, чей тип содержит класс, который не определяется как скалярное свойство (`string` – это класс, но он определяется как скалярное свойство), считаются навигационными. Они могут отображаться как отдельная связь (например, `public PriceOffer Promotion { get; set; }`) или тип, реализующий интерфейс `IEnumerable<T>` (например, `public ICollection<Review> Reviews { get; set; }`).
3. EF Core проверяет, есть ли у каждого из этих классов сущностей первичный ключ (см. раздел 7.9). Если у класса нет первичного ключа и он не был исключен или сконфигурирован как класс, у которого нет ключа (см. раздел 7.9.3), то EF Core выбросит исключение.

8.4.2 Пример класса сущности с навигационными свойствами

В листинге 8.1 показан класс сущности `Book`, определенный в DbContext. В этом случае у нас есть открытое свойство типа `DbSet<Book>`

с классом, который прошел тест «должен иметь действительный первичный ключ», так как у него есть открытое свойство `BookId`.

Нас интересует, как конфигурация «По соглашению» обрабатывает три навигационных свойства в нижней части класса. Как будет показано в этом разделе, EF Core может определить, какие это связи, по типу навигационного свойства и внешнего ключа в классе, на который ссылается навигационное свойство.

Листинг 8.1 Класс сущности Book и связи, указанные в нижней части

```
public class Book
{
    public int BookId { get; set; }
    // Остальные скалярные свойства удалены,
       поскольку не являются релевантными …

    public PriceOffer Promotion { get; set; }

    public ICollection<Tag> Tags { get; set; }

    public ICollection<BookAuthor> AuthorsLink { get; set; }

    public ICollection<Review> Reviews { get; set; }
}
```

Ссылки непосредственно на список сущностей Tag с использованием автоматической связи EF Core 5 «многие ко многим»

Ссылки на PriceOffer. Представляет собой связь «один к нулю или одному»

Ссылки на любые отзывы на эту книгу: связь «один ко многим»

Ссылки на одну сторону связи «многие ко многим» через связующую таблицу

Если между двумя классами сущности есть два навигационных свойства, то эта связь известна как *полностью определенная*, и EF Core может решить, какой это тип: «один к одному» или «один ко многим». Если навигационное свойство только одно, то EF Core не может быть уверен, поэтому предполагает, что это связь «один ко многим».

Некоторым связям «один к одному» может потребоваться настройка через Fluent API, если у вас только одно навигационное свойство или вы хотите изменить настройку «По соглашению» по умолчанию, например когда удаляете класс сущности со связью.

8.4.3 Как EF Core находит внешние ключи по соглашению

Внешний ключ должен соответствовать основному ключу (определенному в разделе 8.1) по типу и имени, но для обработки нескольких сценариев сопоставление имени внешнего ключа имеет три варианта, как показано на рис. 8.2. Здесь видны все три варианта имени внешнего ключа с использованием класса сущности `Review`, который ссылается на первичный ключ `BookId` в классе сущности `Book`.

Вариант 1 я использую чаще всего; он изображен на рис. 8.1. Вариант 2 подходит для разработчиков, которые используют короткое по соглашению имя первичного ключа `Id`, поскольку оно делает внешний ключ уникальным для класса, к которому он привязан. Вариант 3 помогает в конкретных случаях, когда вы получаете повторяющиеся

именованные свойства, если использовали вариант 1. В следующем листинге показан пример использования варианта 3 для управления иерархическими связями.

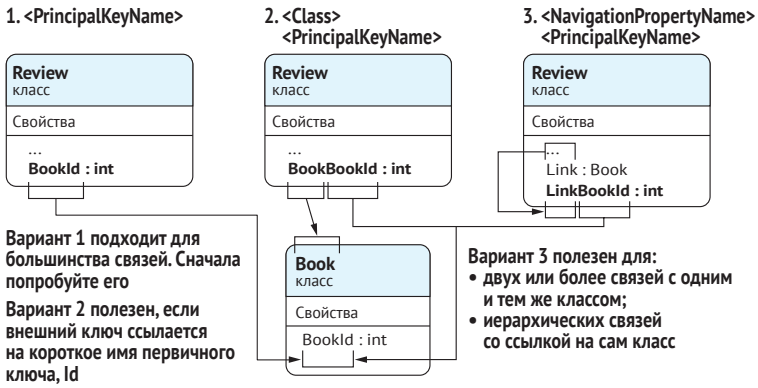


Рис. 8.2 Три варианта «По соглашению» для внешнего ключа, который ссылается на первичный ключ класса сущности Book. Эти варианты позволяют использовать уникальное имя для внешнего ключа, по которому EF Core может определить, к какому первичному ключу относится эта связь

Листинг 8.2 Иерархическая связь с внешним ключом из варианта 3

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    //-----------------------------
    //Связи;

    public int? ManagerEmployeeId { get; set; }
    public Employee Manager { get; set; }
}
```

Внешний ключ использует паттерн <NavigationalPropertyName> <PrimaryKeyName>

У класса сущности `Employee` есть навигационное свойство `Manager`, которое связывает руководителя сотрудника, который также является сотрудником. Внешний ключ `EmployeeId` (вариант 1) использовать нельзя, поскольку он уже используется для первичного ключа. Следовательно, вы используете вариант 3 и вызываете внешний ключ `ManagerEmployeeId`, используя имя навигационного свойства в начале.

8.4.4 Поддержка значения null у внешних ключей: обязательные или необязательные зависимые связи

Допустимость значения null внешнего ключа определяет, является связь обязательной (внешний ключ не допускает значения null) или нет (внешний ключ, допускающий значение null). *Обязательная связь*

гарантирует, что эти связи существуют, гарантируя, что внешний ключ связан с действительным основным ключом. В разделе 8.6.1 описывается сущность `Attendee`, у которой есть обязательная связь с классом сущности `Ticket`.

Необязательная связь допускает отсутствие связи между основной сущностью и зависимой, задав для внешнего ключа значение (значения) null. Навигационное свойство `Manager` в классе сущности `Employee`, показанное в листинге 8.2, – пример необязательной связи, поскольку у кого-то на вершине бизнес-иерархии не будет начальника.

Обязательный или необязательный статус связи также влияет на то, что происходит с зависимыми сущностями при удалении основной сущности. По умолчанию для действия `OnDelete` для каждого типа связи установлено следующее:

- в случае с обязательной связью EF Core устанавливает для действия `OnDelete` значение `Cascade`. Если основная сущность удаляется, зависимая сущность также будет удалена;
- в случае с необязательной связью EF Core устанавливает для действия `OnDelete` значение `ClientSetNull`. Если зависимая сущность отслеживается, то при удалении основной сущности для внешнего ключа будет установлено значение `null`. Но если она *не* отслеживается, то в силу вступает параметр удаления ограничения базы данных, а параметр `ClientSetNull` устанавливает правила базы данных, как если бы параметр `Restrict` был на месте. В результате на уровне базы данных удаление не выполняется и выбрасывается исключение.

ПРИМЕЧАНИЕ Поведение `ClientSetNull` довольно необычно, и в разделе 8.8.1 объясняется, почему. Там также рассказывается, как настроить поведение удаления связи.

8.4.5 Внешние ключи: что произойдет, если не указать их?

Если EF Core находит связь через навигационное свойство или связь, которую вы настроили через Fluent API, ей требуется внешний ключ для настройки связи в реляционной базе данных. Включение внешних ключей в классы сущностей – хорошая практика. Она позволяет лучше контролировать допустимость значения null у внешнего ключа. Кроме того, доступ к внешним ключам может быть полезен при обработке связей в отключенном состоянии (см. раздел 3.3.1).

Но если вы не укажете внешний ключ (намеренно или случайно), то конфигурация EF Core добавит внешний ключ как теневое свойство. *Теневые свойства*, которые описывались в главе 7, – это скрытые свойства, к которым можно получить доступ только с помощью конкретных команд EF Core. Автоматическое добавление внешних ключей в качестве теневых свойств может быть полезным. У одного из моих клиентов, например, был универсальный класс сущности `Note`, который был добавлен в коллекцию `Notes` во многих сущностях.

На рис. 8.3 показана связь «один ко многим», в которой используется класс сущности `Note` в навигационном свойстве коллекции в двух классах сущности: `Customer` и `Job`. Обратите внимание, что имена первичных ключей этих классов используют разные подходы к именованию по соглашению, чтобы показать, как именуются теневые свойства.

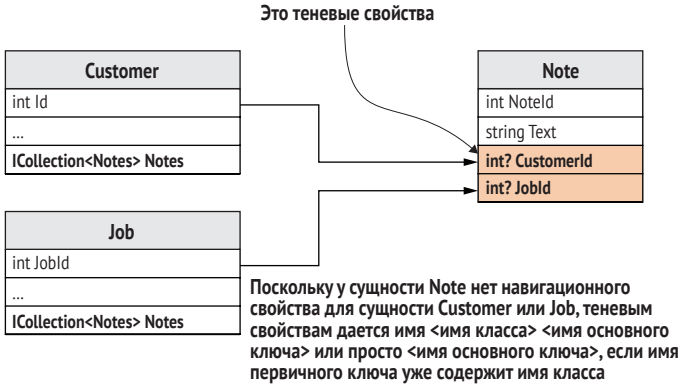


Рис. 8.3 Конфигурация «По соглашению» добавит внешние ключи, допускающие значения null (т. е. необязательная связь) как теневые свойства, если вы не предоставите собственные внешние ключи в классе сущности `Note`

Если у класса сущности, который получает внешний ключ теневого свойства, есть навигационная ссылка на другой конец связи, имя этого теневого свойства будет иметь вид `<имя навигационного свойства> <имя свойства основного ключа>`. Если у сущности `Note` на рис. 8.3 есть навигационная ссылка на сущность `Customer`, `LinkBack`, имя внешнего ключа теневого свойства будет `LinkBackId`.

ПРИМЕЧАНИЕ Мои модульные тесты показывают, что связи «один к одному» отклоняются, если нет внешнего ключа для связи двух сущностей. Таким образом, EF Core по соглашению не будет автоматически настраивать внешние ключи теневого свойства для связей «один к одному».

Если вы хотите добавить внешний ключ в качестве теневого свойства, то можете сделать это, используя метод Fluent API `HasForeignKey`, показанный в разделе 8.6. Имя теневого свойства предоставляется как строка. Будьте осторожны: не используйте имя существующего свойства. Так вы не добавите теневое свойство, а будете использовать существующее.

Теневое свойство внешнего ключа будет допускать значение null. Это эффект, описанный в разделе 8.4.4 о допустимости пустых значений внешних ключей. Если он вам не нравится, то можно изменить допустимость пустых значений теневого свойства с помощью метода Fluent API `IsRequired`, как описано в разделе 8.8.2.

EF6 EF6.x использует аналогичный подход к добавлению внешних ключей, если вы не указали их в своих классах сущности, но в EF6.x нельзя настроить допустимость значений null или получить доступ к содержимому. Теневые свойства EF Core делают ситуацию с недобавлением явных внешних ключей более управляемой.

8.4.6 Когда подход «По соглашению» не работает?

Если вы собираетесь использовать подход «По соглашению», вам необходимо знать, когда он не работает, чтобы можно было использовать другие средства для настройки связи. Вот мой список сценариев, которые не сработают. Наиболее распространенные указаны первыми:

- у вас есть составные внешние ключи (см. раздел 8.6 или 8.5.1);
- вы хотите создать связь «один к одному» без навигационных ссылок в обе стороны (см. раздел 8.6.1);
- вы хотите переопределить поведение удаления по умолчанию (см. раздел 8.8.1);
- у вас есть два навигационных свойства, относящихся к одному и тому же классу (см. раздел 8.5.2);
- вы хотите определить конкретное ограничение базы данных (см. раздел 8.8.4).

8.5 Настройка связей с помощью аннотаций данных

Только две аннотации – `ForeignKey` и `InverseProperty` – относятся к связям, поскольку бóльшая часть навигационной конфигурации выполняется через Fluent API.

8.5.1 Аннотация ForeignKey

Аннотация `ForeignKey` позволяет определить внешний ключ для навигационного свойства в классе. Возьмем иерархический пример класса `Employee`. Можно использовать эту аннотацию для определения внешнего ключа для навигационного свойства `Manager`. В следующем листинге показан обновленный класс сущности `Employee` с новым, более коротким именем внешнего ключа для навигационного свойства `Manager`, которое не соответствует именованию «По соглашению»: `ManagerEmployeeId`.

Листинг 8.3 Использование аннотации ForeignKey для настройки имени внешнего ключа

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }

    public int? ManagerId { get; set; }
    [ForeignKey(nameof(ManagerId))]
    Employee Manager { get; set; }
}
```

Определяет, какое свойство является внешним ключом для навигационного свойства Manager

ПРИМЕЧАНИЕ Мы применили аннотацию данных `ForeignKey` к навигационному свойству, дав имя внешнему ключу `ManagerId`. Но эта аннотация может работать и наоборот. Можно было бы применить ее к свойству внешнего ключа, `ManagerId`, чтобы дать имя навигационному свойству, `Manager`, например `[ForeignKey(nameof(Manager))]`.

Аннотация данных `ForeignKey` принимает один параметр, строку. Эта строка должна содержать имя свойства внешнего ключа. Если внешний ключ является составным (у него несколько свойств), то он должен быть разделен запятыми – например, `[ForeignKey("Property1, Property2")]`.

СОВЕТ Я предлагаю использовать ключевое слово `nameof` для этой строки. Это безопаснее, потому что если вы измените имя свойства внешнего ключа, `nameof` будет обновлено одновременно, либо выдаст ошибку компиляции, если вы забудете поменять все ссылки.

8.5.2 Аннотация InverseProperty

`InverseProperty` – узкоспециализированная аннотация, которая используется, когда у вас есть два навигационных свойства, относящихся к одному и тому же классу. В этом случае EF Core не может определить, какие внешние ключи с каким навигационным свойством связаны. Эту ситуацию лучше всего продемонстрировать в коде. В следующем листинге показан класс сущности `Person` с двумя списками: один для книг, принадлежащих библиотекарю, а другой для книг, которые одолжили конкретному человеку.

Листинг 8.4 Класс сущности LibraryBook с двумя связями с классом Person

```
public class LibraryBook
{
    public int LibraryBookId { get; set; }
```

```
    public string Title { get; set; }

    public int LibrarianPersonId { get; set; }
    public Person Librarian { get; set; }

    public int? OnLoanToPersonId { get; set; }
    public Person OnLoanTo { get; set; }
}
```

`Librarian` и лицо, одолжившее книгу (навигационное свойство `OnLoanTo`), представлены классом сущности `Person`. Навигационные свойства `Librarian` и `OnLoanTo` связаны с одним и тем же классом, и EF Core не может установить навигационную ссылку без посторонней помощи. Аннотация `InverseProperty` в следующем листинге представляет информацию EF Core при настройке навигационных ссылок.

Листинг 8.5 Класс сущности `Person`, использующий аннотацию `InverseProperty`

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }

    [InverseProperty("Librarian")]
    public ICollection<LibraryBook>
        LibrarianBooks { get; set; }

    [InverseProperty("OnLoanTo")]
    public ICollection<LibraryBook>
        BooksBorrowedByMe { get; set; }
}
```

Связывает LibrarianBooks с навигационным свойством Librarian в классе LibraryBook

Связывает список BooksBorrowedByMe с навигационным свойством OnLoanTo в классе LibraryBook

Этот код – один из тех параметров конфигурации, которые редко используются, но если вы находитесь в такой ситуации, то должны либо использовать его, либо определить связь через Fluent API. Иначе EF Core выдаст исключение при запуске, поскольку не сможет понять, как настроить связи.

8.6 Команды Fluent API для настройки связей

Как было сказано в разделе 8.4, можно настроить бóльшую часть связей с помощью подхода «По соглашению». Но если вы хотите настроить связь, то у Fluent API есть хорошо продуманный набор команд, охватывающий все возможные комбинации связей. Кроме того, у него есть дополнительные команды, позволяющие определять ограничения других баз данных. На рис. 8.4 показан формат определения связей через Fluent API. Все команды конфигурации связей Fluent API следуют этому шаблону.

EF6 Имена команд Fluent API не такие, как в EF6, и, как по мне, они намного понятнее. Команды EF6 `WithRequired` и `WithRequiredPrincipal/WithRequiredDependent` мне казались немного запутанными, тогда как команды Fluent API имеют более четкий синтаксис `HasOne/HasMany`, за которым следует `WithOne/WithMany`.

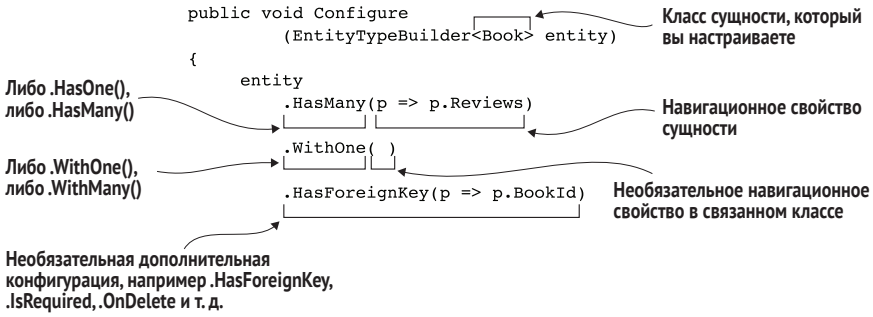


Рис. 8.4 Fluent API позволяет определять связи между двумя классами сущности. `HasOne/HasMany` и `WithOne/WithMany` – две основные части, за которыми следуют другие команды, чтобы указать другие части или настроить определенные функции

Далее мы определим связи «один к одному», «один ко многим» и «многие ко многим», чтобы проиллюстрировать, как использовать их с Fluent API.

8.6.1 Создание связи «один к одному»

Связи «один к одному» могут быть немного сложными, потому что создать их в реляционной базе данных можно тремя способами. Чтобы разобраться с этими вариантами, мы рассмотрим пример, в котором у нас есть участники (класс сущности `Attendee`) конференции по программному обеспечению и у каждого участника есть уникальный билет (класс сущности `Ticket`).

В главе 3 показано, как создавать, обновлять и удалять связи. Вот фрагмент кода, где показано, как создать связь «один к одному»:

```
var attendee = new Attendee
{
    Name = "Person1",
    Ticket = new Ticket{ TicketType = TicketTypes.VIP}
};
context.Add(attendee);
context.SaveChanges();
```

На рис. 8.5 представлены три варианта построения такого рода связей. Основные сущности находятся в верхней части диаграммы, а зависимые сущности – внизу. Обратите внимание, что в варианте 1

`Attendee` – это зависимая сущность, а в вариантах 2 и 3 зависимая сущность – это `Ticket`.

У каждого варианта есть свои достоинства и недостатки. Вам следует использовать тот, который подходит для ваших бизнес-потребностей.

Вариант 1 – стандартный подход к созданию связей «один к одному», потому что он позволяет определить, что зависимая сущность «один к одному» обязательна (значение не должно быть равно null).

В нашем примере будет сгенерировано исключение, если вы попытаетесь сохранить экземпляр сущности `Attendee` без прикрепленного к ней уникального билета. На рис. 8.6 этот вариант показан более подробно.

Используя вариант 1, можно сделать зависимую сущность необязательной, сделав так, что внешний ключ будет поддерживать значение null. Кроме того, на рис. 8.6 видно, что у метода `WithOne` есть параметр, выбирающий навигационное свойство `Attendee` в классе сущности `Ticket`, которое ссылается на класс сущности `Attendee`.

Поскольку класс `Attendee` является зависимой частью связи, если вы удалите сущность `Attendee`, связанный билет не будет удален, поскольку `Ticket` – это основная сущность в связи. Обратная сторона варианта 1 в этом примере состоит в том, что он позволяет использовать один билет для нескольких участников, а это не соответствует бизнес-правилам, о которых я говорил в начале. Наконец, этот вариант позволяет заменить `Ticket` другим экземпляром `Ticket`, назначив новый билет навигационному свойству посетителя `Ticket`.

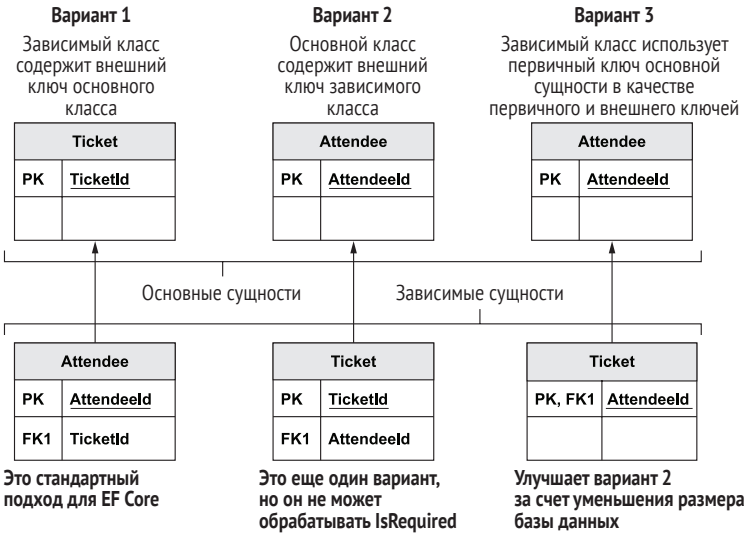


Рис. 8.5 Три способа определения связи «один к одному» в реляционной базе данных; комментарии внизу указывают на то, как EF Core работает с каждым подходом. Вариант 1 отличается от вариантов 2 и 3 тем, что концы связи меняются местами. Это меняет то, какую часть можно заставить существовать. В варианте 1 у участника должен быть билет, тогда как в вариантах 2 и 3 билет не является обязательным. Кроме того, если основная сущность (верхняя строка) удалена, зависимая сущность (нижняя строка) тоже будет удалена

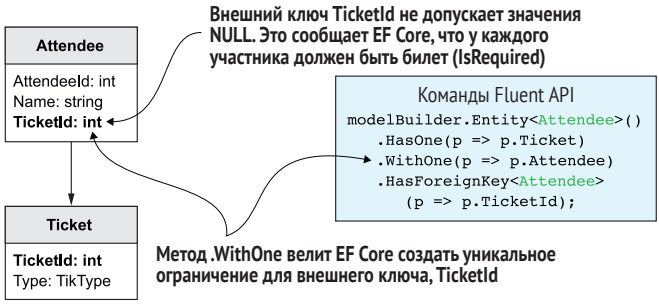


Рис. 8.6 Внешний ключ, не допускающий значения null, гарантирует, что у основной сущности (в данном случае `Attendee`) должна быть зависимая сущность «один к одному», `Ticket`. Кроме того, настройка связи по типу «один к одному» гарантирует, что каждая зависимая сущность, `Ticket`, уникальна. Обратите внимание, что у Fluent API справа навигационные свойства идут в обе стороны; у каждой сущности есть навигационное свойство, ведущее к другой

Варианты 2 и 3 на рис. 8.5 изменяют связь между основной и зависимой сущностями. `Attendee` становится основной сущностью. Эта ситуация меняет местами природу связи «обязательная/необязательная». Теперь `Attendee` может существовать без `Ticket`, но `Ticket` не может существовать без `Attendee`. Варианты 2 и 3 принудительно назначают `Ticket` только одному `Attendee`, но чтобы заменить `Ticket` на другой экземпляр `Ticket`, сначала нужно удалить старый билет. Эта связь показана на рис. 8.7.

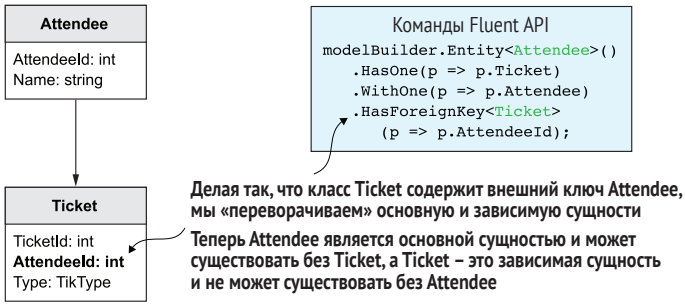


Рис. 8.7 Вариант 2: сущность `Ticket` содержит внешний ключ сущности `Attendee`. Таким образом вы выбираете, какая сущность является основной, а какая – зависимой. В данном случае `Attendee` теперь является основной сущностью, а `Ticket` – зависимой

Варианты 2 и 3 полезны, потому что они образуют необязательные связи «один к одному», которые часто называют связями «один к нулю или к одному». Вариант 3 – более эффективный способ определения варианта 2 с объединением первичного и внешнего ключей. Я бы использовал вариант 3 для класса сущности `PriceOffer` в приложении Book App, но мне хотелось начать с более простого подхода – варианта 2. В другой, даже еще лучшей версии используется тип

Owned (см. раздел 8.9.1), потому что он автоматически загружается из той же таблицы. Это безопаснее (я не забуду добавить метод Include) и более эффективно.

8.6.2 Создание связи «один ко многим»

Связи «один ко многим» проще, потому что здесь существует один формат: множество сущностей содержат значение внешнего ключа. Можно определить большинство таких связей с помощью подхода «По соглашению», просто задав внешнему ключу во многих сущностях имя, которое следует этому подходу (см. раздел 8.4.3). Но если вы хотите определить связь, то можно использовать Fluent API, который полностью контролирует настройку связей. На рис. 8.8 представлен пример кода Fluent API для создания связи «у одной сущности Book много Review» в приложении Book App.

В этом случае у класса сущности Review нет навигационной ссылки на Book, поэтому у метода WithOne нет параметра.

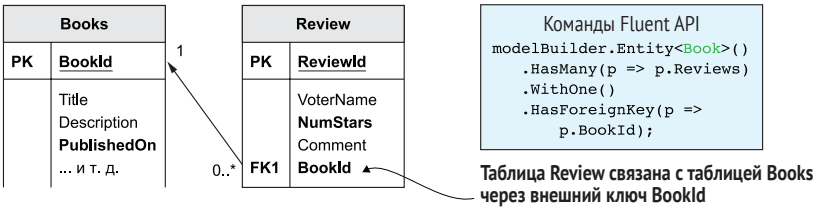


Рис. 8.8 Связь «один ко многим», в которой внешний ключ должен находиться в зависимой сущности – в данном случае в классе сущности Review. В Fluent API справа видно, что у Book есть навигационное свойство коллекции Reviews, связанное с классами сущности Review, но у Review нет навигационного свойства для Book

ПРИМЕЧАНИЕ В листинге 3.16 показано, как добавить Review в навигационное свойство класса Book, коллекцию Reviews.

У коллекций есть пара особенностей, о которых стоит знать. Во-первых, вы можете использовать любой обобщенный тип для коллекции, реализующий интерфейс IEnumerable<T>, например IList<T>, Collection<T>, HashSet<T>, List<T> и др. IEnumerable<T> сам по себе – особый случай, поскольку в него нельзя добавить элементы.

По соображениям производительности для навигационных коллекций нужно использовать HashSet<T>, потому что это улучшает определенные части процессов запросов и обновления в EF Core. (См. главу 14 для получения дополнительной информации по этой теме.) Но HashSet не гарантирует порядок записей, а это может вызвать проблемы, если использовать сортировку в методах Include (см. раздел 2.4.1, листинг 2.5). Вот почему в частях I и II я рекомендую использовать ICollection<T>, если вы можете отсортировать методы In-

clude, поскольку ICollection сохраняет порядок, в котором добавляются записи. Но в части III, касающейся производительности, мы не используем сортировку в методах Include, поэтому для повышения производительности можно использовать HashSet<T>.

Во-вторых, хотя обычно навигационное свойство коллекции определяется с методами чтения и записи (например, public ICollection<Review> Reviews {get; set;}), в этом нет необходимости. Вы можете предоставить только метод чтения, если инициализируете резервное поле с пустой коллекцией. Приведенный ниже код также является допустимым:

```
public ICollection<Review> Reviews { get; } = new List<Review>();
```

Хотя в этом случае инициализация коллекции может упростить задачу, я не рекомендую инициализировать навигационное свойство коллекции. Причины я объяснил в разделе 6.1.6.

8.6.3 Создание связей «многие ко многим»

Связи «многие ко многим» описаны в главах 2 и 3; в этом разделе вы узнаете, как их настроить. В главах 2 и 3 мы познакомились с двумя типами связей «многие ко многим»:

- *ваша связующая таблица содержит информацию, к которой вы хотите получить доступ при чтении данных на другой стороне связи «многие ко многим».* Примером может служить связь «многие ко многим» между Book и Author, где связующая таблица содержит порядок, в котором должны отображаться имена авторов;
- *вы напрямую получаете доступ к другой стороне связи «многие ко многим».* Примером может служить связь «многие ко многим» между Book и Tags, где можно получить прямой доступ к коллекции Tags в классе сущности Book без необходимости доступа к связующей таблице.

КОНФИГУРИРОВАНИЕ СВЯЗИ «МНОГИЕ КО МНОГИМ» С ИСПОЛЬЗОВАНИЕМ СВЯЗУЮЩЕГО КЛАССА СУЩНОСТИ

Мы начнем со связи «многие ко многим», где вы получаете доступ к другому концу связи через связующую таблицу. Эта связь требует больше работы, однако она позволяет добавлять дополнительные данные в связующую таблицу, по которым можно сортировать или фильтровать. Как это сделать, было показано в разделе 3.4.4. На рис. 8.9 видны фрагменты конфигурации этой связи.

В примере Book/Author, используя подход «По соглашению», можно найти и связать все скалярные и навигационные свойства, поэтому единственная необходимая конфигурация – это настройка первичного ключа. В следующем фрагменте кода используется Fluent API в методе DbContext OnModelCreating:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     modelBuilder.Entity<BookAuthor>()
           .HasKey(x => new {x.BookId, x.AuthorId});
}
```

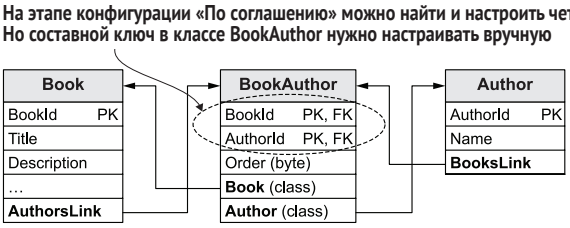


Рис. 8.9 Три класса сущностей, участвующих в связи «многие ко многим», где используется связующая таблица. Этот тип связи применяется только в том случае, если у вас есть дополнительные данные в классе сущности связующей таблицы. В этом случае класс `BookAuthor` содержит свойство `Order`, определяющее порядок, в котором должны быть указаны имена авторов, отображаемые наряду с `Book`

Можно настроить четыре связи в связи «многие ко многим», используя Fluent API с кодом из следующего листинга. Обратите внимание, что команды Fluent API `HasOne/WithMany` из листинга не обязательны, потому что класс сущности `BookAuthor` следует правилам именования и ввода «По соглашению».

Листинг 8.6 Настройка связи «многие ко многим», где используются две связи «один ко многим»

```
public static void Configure
    (this EntityTypeBuilder<BookAuthor> entity)
{
    entity.HasKey(p =>
         new { p.BookId, p.AuthorId });

    //----------------------------
    //Связи;

    entity.HasOne(p => p.Book)
        .WithMany(p => p.AuthorsLink)
        .HasForeignKey(p => p.BookId);

    entity.HasOne(p => p.Author)
        .WithMany(p => p.BooksLink)
        .HasForeignKey(p => p.AuthorId);
}
```

Использует имена первичных ключей Book и Author для формирования собственного составного ключа

Настраивает связь «один ко многим» для класса сущности BookAuthor и Book

Настраивает связь «один ко многим» для класса сущности BookAuthor и Author

НАСТРОЙКА СВЯЗИ «МНОГИЕ КО МНОГИМ» С ПРЯМЫМ ДОСТУПОМ К ДРУГОЙ СУЩНОСТИ

После выхода EF Core 5 можно ссылаться на другой конец связи «многие ко многим» напрямую. Пример, показанный в главах 2 и 3, – класс сущности `Book`, у которого есть навигационное свойство `ICollection<Tag> Tags`, содержащее серию классов сущности `Tag`. Класс сущности `Tag` содержит категорию (`Microsoft .NET`, `Web` и т. д.), что помогает покупателю найти нужную книгу.

Конфигурация «По соглашению» хорошо подходит для прямой связи «многие ко многим». Если классы сущностей на двух концах являются допустимыми, то конфигурация «По соглашению» установит связи и ключи за вас, как показано на рис. 8.10, а также создаст для вас связующую сущность, используя «контейнер свойств» (см. раздел 8.9.5).

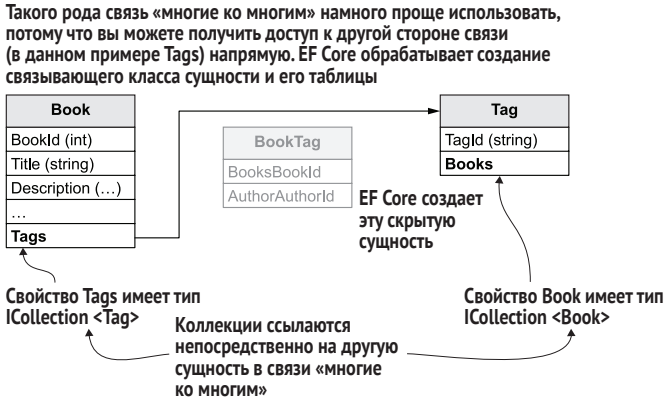


Рис. 8.10 Прямая связь «многие ко многим» в EF Core 5 работает, потому что (а) EF Core создает за вас связующий класс сущности и (б) когда он видит запрос, содержащий прямую связь «многие ко многим», то добавляет команды SQL для использования скрытого связующего класса сущности. Отсутствие необходимости создавать связующий класс сущности или выполнять конфигурирование намного упрощает настройку таких связей

Но если вы хотите добавить собственную связующую таблицу и конфигурацию, это можно сделать, используя Fluent API. Класс сущности для связующей таблицы аналогичен классу сущности `BookAuthor`, показанному на рис. 8.9. Разница состоит в том, что ключ/связь `Author` заменяется ключом/связью `Tag`. В следующем листинге показан класс конфигурации `Book`, настраивающий класс сущности `BookTag` для связывания двух частей.

Листинг 8.7 Настройка прямых связей «многие ко многим» с использованием Fluent API

```
public void Configure
    (EntityTypeBuilder<Book> entity)
{
    //… Другие конфигурации опущены для ясности;

    entity.HasMany(x => x.Tags)
        .WithMany(x => x.Books)
        .UsingEntity<BookTag>(
            bookTag => bookTag.HasOne(x => x.Tag)
                .WithMany().HasForeignKey(x => x.TagId),
            bookTag => bookTag.HasOne(x => x.Book)
                .WithMany().HasForeignKey(x => x.BookId));
}
```

HasMany/WithMany устанавливает прямую связь «многие ко многим»

Метод UsingEntity<T> позволяет определить класс сущности для связующей таблицы

Определение стороны Tag связи «многие ко многим»

Определение стороны Book связи «многие ко многим»

Код, показанный в листинге 8.7, не делает ничего, кроме замены связующей сущности, который добавила бы EF Core, так что не стоит его использовать. Но он был бы полезен, если бы вы захотели добавить дополнительные свойства класса сущности `BookTag`, например свойство `SoftDeleted`, использующее фильтр запросов для мягкого удаления ссылки.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ После выхода EF Core 5 появилось полезное видео, где рассказывается о связях «многие ко многим» с прямым доступом, включая добавление собственных связующих таблиц (подходы TPH и TPT). См. http://mng.bz/opzM.

8.7 Управление обновлениями навигационных свойств коллекции

Иногда нужно контролировать доступ к навигационным свойствам коллекции. Несмотря на то что вы можете контролировать доступ к навигационным свойствам «один к одному», делая метод записи закрытым, данный подход не работает для коллекции, поскольку большинство типов коллекций позволяют добавлять или удалять записи. Чтобы полностью управлять навигационными свойствами коллекции, необходимо использовать резервные поля, описанные в разделе 7.14.

EF6.X У EF6.x не было возможности контролировать доступ к навигационным свойствам коллекции. Это означало, что некоторые подходы, например предметно-ориентированное проектирование, было бы непросто реализовать. Резервные

поля EF Core позволяют создавать классы сущностей, соответствующие принципам предметно-ориентированного проектирования.

Сохранение коллекции связанных классов сущности в поле позволяет перехватить любую попытку обновить коллекцию. Вот несколько причин, по которым эта функция полезна для бизнес-логики и проектирования программного обеспечения:

- запуск бизнес-логики при изменении, например вызов метода, если коллекция содержит более десяти записей;
- создание локального кешированного значения из соображений производительности, например для хранения кешированного свойства `ReviewsAverageVotes` всякий раз, когда `Review` добавляется в класс сущности `Book` или удаляется из него;
- применение предметно-ориентированного проектирования к классам сущностей. Любое изменение данных должно производиться с помощью метода (см. главу 13).

В качестве примера управления навигационными свойствами коллекции мы добавим кешированное свойство `ReviewsAverageVotes` в класс `Book`. Оно будет содержать среднюю оценку из отзывов, связанных с этой книгой. Для этого необходимо:

- добавить резервное поле `_reviews` для хранения коллекции `Reviews` и изменить свойство, чтобы возвращать копию коллекции с доступом только на чтение, содержащейся в поле `_reviews`;
- добавить свойство с доступом только на чтение `ReviewsAverageVotes`, чтобы хранить там кешированное среднее значение оценки из `Reviews`, связанных с этой сущностью `Book`;
- добавить методы для добавления и удаления отзывов из поля `_reviews`. Каждый метод будет пересчитывать среднюю оценку, используя текущий список `Reviews`.

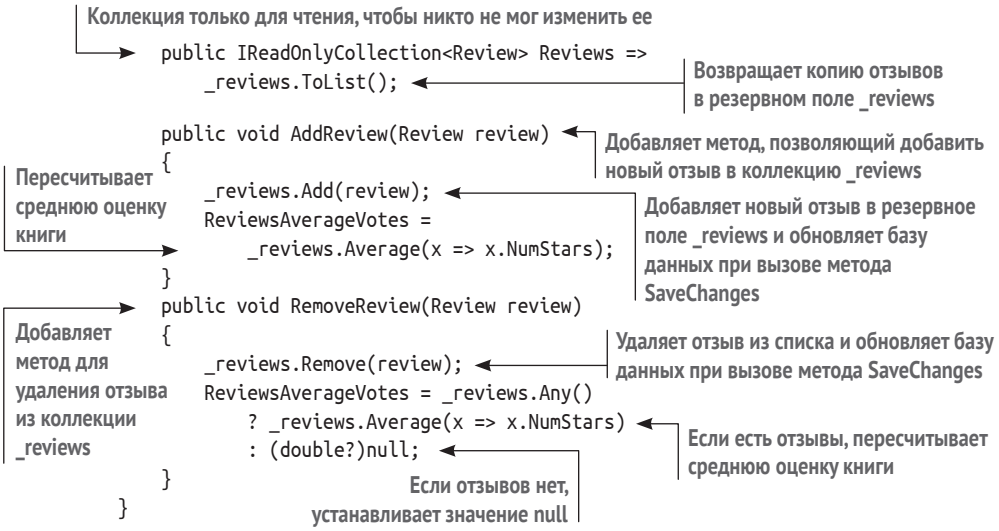
В следующем листинге показан обновленный класс `Book`. Здесь показан код, относящийся к отзывам и кешированному свойству `ReviewsAverageVotes`.

Листинг 8.8 Класс Book с навигационным свойством коллекции Reviews с доступом только на чтение

```
public class Book
{
    private readonly ICollection<Review> _reviews
        = new List<Review>();

    public int BookId { get; set; }
    public string Title { get; set; }
    //… Другие свойства и связи не указаны;

    public double? ReviewsAverageVotes { get; private set; }
```

Вы добавляете резервное поле, представляющее собой список. По умолчанию EF Core будет читать и писать данные в это поле

Содержит предварительно вычисленное среднее значение оценки с доступом только на чтение



Не нужно настраивать резервное поле, потому что мы использовали именование «По соглашению», и по умолчанию EF Core считывает и записывает данные в поле `_reviews`.

В этом примере показано, как сделать навигационные свойства коллекции доступными только для чтения, но это не идеальный вариант, потому что одновременные обновления могут сделать свойство `ReviewsAverageVotes` неактуальным. В части III мы создадим приложение, используя предметно-ориентированное проектирование, и реализуем надежный подход к кешированию, который решает проблемы параллельного доступа.

8.8 Дополнительные методы, доступные во Fluent API

Мы рассмотрели все способы настройки стандартных связей, но некоторые связи нуждаются в более детальных настройках и требуют добавления дополнительных команд в конфигурацию Fluent API. В этом разделе мы рассмотрим четыре метода, определяющих более глубокие аспекты связей:

- `OnDelete` – изменяет действие при удалении зависимой сущности;
- `IsRequired` – определяет допустимость пустых значений внешнего ключа;
- `HasPrincipalKey` – использует альтернативный уникальный ключ;
- `HasConstraintName` – устанавливает имя ограничения по внешнему ключу и свойство `MetaData` для доступа к данным связей.

8.8.1 OnDelete: изменение действия при удалении зависимой сущности

В разделе 8.4.4 описывается действие по умолчанию при удалении основной сущности, которое основано на допустимости пустых значений внешнего ключа (ключей) зависимой сущности. Метод Fluent API `OnDelete` позволяет изменить действия EF Core, когда удаление затрагивает зависимую сущность.

Можно добавить метод `OnDelete` в конец конфигурации Fluent API. В этом листинге показан код, добавленный в главе 4, чтобы предотвратить удаление сущности `Book`, если она упоминается в заказе, через класс сущности `LineItem`.

Листинг 8.9 Изменение действия `OnDelete` по умолчанию для зависимой сущности

```
public static void Configure
    (this EntityTypeBuilder<LineItem> entity)
{
    entity.HasOne(p => p.ChosenBook)
      .WithMany()
      .OnDelete(DeleteBehavior.Restrict);
}
```

Добавляет метод OnDelete в конец определения связи

Этот код вызывает исключение, если кто-то пытается удалить сущность `Book`, на которую ссылается внешний ключ `LineItem`. Вы делаете это, потому что не хотите, чтобы заказ покупателя изменялся. В табл. 8.1 приводится объяснение возможных параметров `DeleteBehavior`.

Таблица 8.1 Действия при удалении, доступные в EF Core. В среднем столбце выделено поведение при удалении, которое будет использоваться, если не применить параметр `OnDelete`

Имя	Влияние удаления на зависимую сущность	По умолчанию для
`Restrict`	Операция удаления не применяется к зависимым сущностям. Зависимые сущности остаются неизменными, что может привести к сбою удаления в EF Core или в реляционной базе данных	
`SetNull`	Зависимая сущность не удаляется, но для ее свойства внешнего ключа устанавливается значение `null`. Если какое-либо из свойств внешнего ключа зависимой сущности не допускает значения null, возникает исключение при вызове метода `SaveChanges`	
`ClientSetNull`	Если EF Core отслеживает зависимую сущность, то для ее внешнего ключа устанавливается значение `null`, и зависимая сущность не удаляется. Но если EF Core не отслеживает зависимую сущность, применяются правила базы данных. В базе данных, созданной EF Core, `DeleteBehavior` установит для ограничения SQL DELETE значение NO ACTION, что вызовет сбой удаления и генерацию исключения	Необязательные связи
`Cascade`	Зависимая сущность удаляется	Обязательные связи

Таблица 8.1 (*окончание*)

Имя	Влияние удаления на зависимую сущность	По умолчанию для
`ClientCascade`	Для сущностей, отслеживаемых DbContext, зависимые сущности будут удалены при удалении связанной основной сущности. Но если EF Core не отслеживает зависимую сущность, применяются правила базы данных. В базе данных, созданной EF Core, для этого будет установлено значение `Restrict`, что приведет к сбою удаления и генерации исключения	

Здесь есть два действия, чьи имена начинаются с `Client`: `ClientSetNull` (добавлен в EF Core 2.0) и `ClientCascade` (добавлен в EF Core 3.0). Они переносят часть операций по удалению из базы данных на клиента, т. е. в код EF Core. Я считаю, что эти два варианта были добавлены для предотвращения проблем, которые могут возникнуть в некоторых базах данных, таких как SQL Server, когда у сущностей есть навигационные ссылки, которые возвращаются сами к себе. В этих случаях вы получите сообщение об ошибке с сервера базы данных, если попытаетесь создать свою базу данных, которую, возможно, будет непросто диагностировать и привести в порядок.

В обоих случаях эти команды выполняют код внутри EF Core, который делает ту же работу, что и база данных с действиями `SetNull` и `Cascade` соответственно. Но – и это большое *но* – EF Core может применить эти изменения, только если вы загрузили все соответствующие зависимые сущности, связанные с основной сущностью, которую вы собираетесь удалить. Если вы этого не сделаете, база данных применит свои правила удаления, что обычно приводит к выбросу исключения.

`ClientSetNull` используется по умолчанию для дополнительных связей, а EF Core установит для внешнего ключа загруженной зависимой сущности значение `null`. Если вы используете EF Core для создания или миграции базы данных, EF Core устанавливает для правил удаления базы данных значение `ON DELETE NO ACTION` (SQL Server). Сервер базы данных не выдаст исключения, если ваши сущности имеют циклические связи (`possible cyclic delete paths` в SQL Server). `SetNull` установит для правил удаления базы данных значение `ON DELETE SET NULL` (SQL Server), что заставит сервер базы данных выбросить исключение `possible cyclic delete paths`.

`ClientCascade` делает то же самое для каскадного удаления в базе данных, поскольку удаляет все загруженные зависимые сущности. Опять же, если вы используете EF Core для создания или миграции базы данных, EF Core устанавливает для правил удаления базы данных значение `ON DELETE NO ACTION` (SQL Server). `Cascade` установит для правил удаления базы данных значение `ON DELETE CASCADE` (SQL Server), что заставит сервер базы данных выбросить исключение `possible cyclic delete paths`.

ПРИМЕЧАНИЕ В документации EF Core есть страница, посвященная каскадному удалению с отработанными примерами;

см. http://mng.bz/nMGK. Кроме того, в ветке Part2 репозитория GitHub есть модульный тест `Ch08_DeleteBehaviour`, содержащий тесты по каждому из вышеописанных параметров.

В листинге 8.10 показан правильный способ использования `ClientSetNull` и `ClientCascade` при удалении основной сущности. Сущность из этого листинга загружается с необязательной зависимой сущностью, где (по умолчанию) используется сценарий `ClientSetNull`. Но тот же код будет работать для `ClientCascade`, если вы загружаете правильную зависимую сущность или сущности.

Листинг 8.10 Удаление основной сущности с необязательной зависимой сущностью

```
var entity = context.DeletePrincipals
    .Include(p => p.DependentDefault)
    .Single(p => p.DeletePrincipalId == 1);

context.Remove(entity);
context.SaveChanges();
```

Читает основную сущность

Включает зависимую сущность, где по умолчанию используется действие ClientSetNull

Настраивает основную сущность для удаления

Вызывает метод SaveChanges, который устанавливает для ее внешнего ключа значение null

Обратите внимание: если вы не включите метод `Include` или другой способ загрузки необязательной зависимой сущности, метод `SaveChanges` вызовет исключение `DbUpdateException`, поскольку сервер базы данных сообщит о нарушении ограничения внешнего ключа. Один из способов согласовать подход EF Core к необязательной связи с подходом сервера базы данных – установить значение `SetNull` вместо `ClientSetNull` по умолчанию, сделав ограничение по внешнему ключу в базе данных `ON DELETE SET NULL` (SQL Server) и возлагая на базу данных ответственность за установку для внешнего ключа значения `null`. Загрузите вы необязательную зависимую сущность или нет, результат вызываемого метода `SaveChanges` будет тем же: внешний ключ необязательной зависимой сущности будет иметь значение `null`.

Но имейте в виду, что некоторые серверы баз данных могут возвращать ошибку при создании базы данных, если у вас есть параметр `SetNull` или `Cascade`, и серверы обнаруживают возможную циклическую связь, например иерархические данные. Вот почему в EF Core есть `ClientSetNull` и `ClientCascade`.

ПРИМЕЧАНИЕ Если вы управляете созданием или миграцией базы данных за пределами EF Core, важно убедиться, что ограничение по внешнему ключу реляционной базы данных соответствует настройке `OnDelete`. В противном случае вы получите непоследовательное поведение, в зависимости от того, отслеживается ли зависимая сущность.

8.8.2 IsRequired: определение допустимости значения null для внешнего ключа

В главе 6 описывается, как метод `IsRequired` в Fluent API позволяет установить допустимость значения null для скалярного свойства, такого как строка. В связях та же команда устанавливает допустимость значения null внешнего ключа, которая, как я уже говорил, определяет, является связь обязательной или нет.

Метод `IsRequired` наиболее полезен в теневых свойствах, поскольку в EF Core по умолчанию теневые свойства допускают значение null, а метод `IsRequired` может изменять их. В следующем листинге показан класс сущности `Attendee`, который мы уже использовали ранее, чтобы показать связь «один к одному». Здесь демонстрируются две другие связи «один к одному», использующие теневые свойства для своих внешних ключей.

Листинг 8.11 Класс сущности `Attendee` и все его связи

```
public class Attendee
{
    public int AttendeeId { get; set; }
    public string Name { get; set; }

    public int TicketId { get; set; }
    public Ticket Ticket { get; set; }

    public MyOptionalTrack Optional { get; set; }
    public MyRequiredTrack Required { get; set; }
}
```

Внешний ключ для связи «один к одному», Ticket

Навигационное свойство, которое обращается к сущности Ticket

Навигационное свойство с использованием теневого свойства для внешнего ключа. Используем команды Fluent API, чтобы сказать, что внешний ключ не допускает значения null, поэтому связь является обязательной

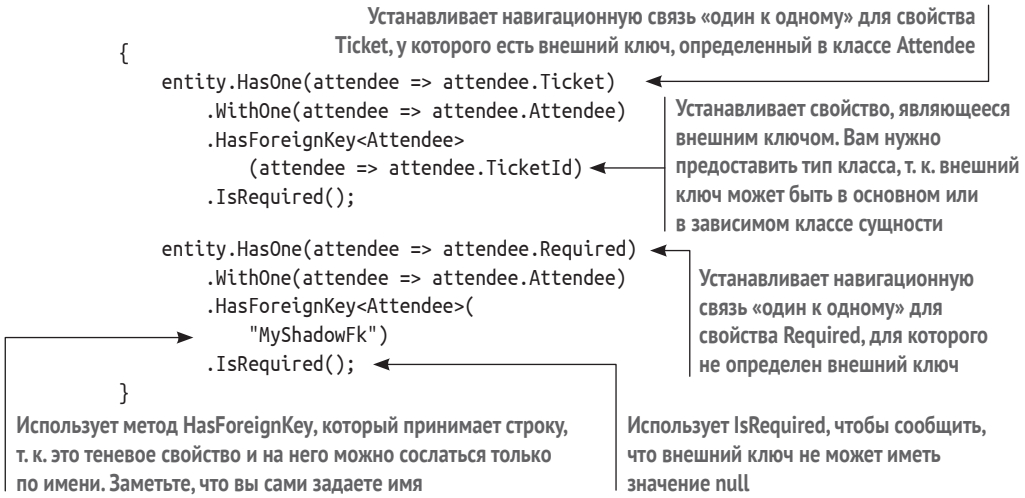
Навигационное свойство, использующее теневое свойство для внешнего ключа. По умолчанию внешний ключ допускает значение null, поэтому связь не является обязательной

Навигационное свойство `Optional`, использующее теневое свойство для своего внешнего ключа, настроено по соглашению. Это означает, что теневое свойство остается значением, допускающим null. Следовательно, оно является необязательным, и если сущность `Attendee` удалена, то сущность `MyOptionalTrack` не удаляется.

Для навигационного свойства `Required` в следующем листинге представлена конфигурация Fluent API. Здесь используется метод `IsRequired`, чтобы сделать навигационное свойство `Required` обязательным. У каждой сущности `Attendee` должна быть сущность `MyRequiredTrack`, присвоенная свойству `Required`.

Листинг 8.12 Конфигурация Fluent API класса сущности `Attendee`

```
public void Configure
    (EntityTypeBuilder<Attendee> entity)
```



Можно было бы опустить конфигурацию навигационного свойства `Ticket`, так как она будет правильно настроена в соответствии с правилами «По соглашению», но мы оставим ее, чтобы можно было сравнить ее с конфигурацией навигационного свойства `Required`, которое использует теневое свойство внешнего ключа. Конфигурация навигационного свойства `Required` необходима, потому что метод `IsRequired` изменяет теневое свойство внешнего ключа с допускающего значение null на значение, не допускающее его, что, в свою очередь, делает связь обязательной.

ТИП И СОГЛАШЕНИЯ ПО ИМЕНОВАНИЮ ВНЕШНИХ КЛЮЧЕЙ С ТЕНЕВЫМИ СВОЙСТВАМИ

Обратите внимание, как листинг 8.12 ссылается на теневое свойство внешнего ключа: нужно использовать метод `HasForeignKey<T>(string)`. Класс `<T>` сообщает EF Core, где разместить теневое свойство внешнего ключа: это может быть либо конец связи «один к одному», либо класс сущности связи «один ко многим».

Строковый параметр метода `HasForeignKey<T>(string)` позволяет определить имя теневого свойства внешнего ключа. Можно использовать любое имя; не нужно придерживаться именования «По соглашению», указанного на рис. 8.3. Но нужно быть осторожным, чтобы не использовать имя существующего свойства из класса сущности, на который вы нацелились, потому что такой подход может привести к странному поведению (если вы все же выберете существующее свойство, то предупреждения не будет, поскольку вы можете попытаться определить нетеневой внешний ключ).

8.8.3 HasPrincipalKey: использование альтернативного уникального ключа

В начале этой главы я упомянул термин *альтернативный ключ*, сказав, что это уникальное значение, но не первичный ключ. Я привел пример альтернативного ключа под названием `UniqueISBN`. Он представляет собой уникальный ключ, который не является первичным (как вы помните, аббревиатура ISBN означает *International Standard Book Number* (международный стандартный книжный номер), представляющий собой уникальный номер для каждой книги).

А теперь рассмотрим другой пример. В следующем листинге создается класс сущности `Person`, который использует обычный первичный ключ типа `int`, но мы будем использовать `UserId` в качестве альтернативного ключа для связи с контактной информацией, показанной в листинге 8.14.

Листинг 8.13 Класс `Person` с `Name`, взятым из авторизации ASP.NET

```
public class Person
       {
    public int PersonId { get; set; }

    public string Name { get; set; }

    public Guid UserId { get; set; }

    public ContactInfo ContactInfo { get; set; }
}
```

Содержит уникальный идентификатор человека

Навигационное свойство, связанное с ContactInfo

Листинг 8.14 Класс `ContactInfo` с `UserIdentifier` в качестве внешнего ключа

```
public class ContactInfo
{
    public int ContactInfoId { get; set; }

    public string MobileNumber { get; set; }
    public string LandlineNumber { get; set; }

    public Guid UserIdentifier { get; set; }
}
```

UserIdentifier используется в качестве внешнего ключа для сущности Person для связи с ContactInfo

На рис. 8.11 показаны команды конфигурации Fluent API, в которых используется альтернативный ключ в классе сущности `Person` в качестве внешнего ключа в классе сущности `ContactInfo`.

Вот несколько примечаний касательно альтернативных ключей:

- у вас могут быть составные альтернативные ключи, состоящие из двух или более свойств. Работают с ними так же, как и с составными ключами: используя анонимный тип, например `HasPrincipalKey<MyClass>(c => new {c.Part1, c.Part2});`

- уникальные (см. раздел 7.10) и альтернативные ключи различаются, и для бизнеса нужно выбрать правильный вариант. Вот некоторые отличия:
  - уникальные ключи гарантируют уникальность каждой записи; их нельзя использовать во внешнем ключе;
  - уникальные ключи могут иметь значение `null`, а альтернативные – нет;
  - значения уникальных ключей можно обновлять, а значения альтернативных ключей – нет (см. http://mng.bz/vzEM);
- можно определить свойство как автономный альтернативный ключ с помощью команды Fluent API `modelBuilder.Entity<Car>().HasAlternateKey(c => c.License-Plate)`, но в этом нет необходимости, потому что, используя метод `HasPrincipalKey` для автоматической установки связи, вы регистрируете свойство как альтернативный ключ.

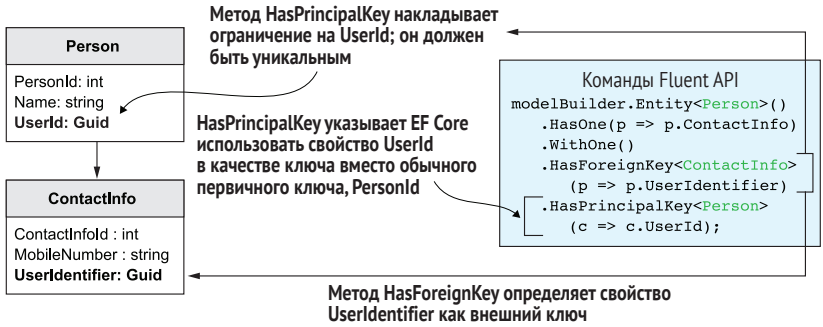


Рис. 8.11 Fluent API устанавливает связь «один к одному» с помощью свойства `UserId`, которое содержит уникальный идентификатор человека в качестве внешнего ключа для связи с `ContactInfo`. Команда `HasPrincipalKey` определяет свойство `UserId` как альтернативный ключ и создает ссылку ограничения по внешнему ключу между свойством `UserIdentifier` в сущности `ContactInfo` и `UserId` в сущности `Person`

8.8.4 Менее используемые параметры в связях Fluent API

В этом разделе кратко упоминаются, но не рассматриваются подробно, две команды Fluent API, которые можно использовать для настройки связей.

HasConstraintName: настройка имени ограничения по внешнему ключу

Метод `HasConstraintName` позволяет установить имя ограничения по внешнему ключу. Это может быть полезно, если вы хотите перехватить исключение при возникновении ошибок внешнего ключа и использовать имя ограничения, чтобы составить более удобное сообщение об ошибке. В этой статье показано, как это сделать: http://mng.bz/4ZwV.

MetaData: ДОСТУП К ИНФОРМАЦИИ О СВЯЗИ

Свойство `MetaData` обеспечивает доступ к данным связи, некоторые из которых предназначены для чтения и записи. Доступ ко многому из того, что предоставляет свойство `MetaData`, можно получить через определенные команды, такие как `IsRequired`, но если вам нужно что-то необычное, посмотрите различные методы и свойства, поддерживаемые свойством `MetaData`.

8.9 Альтернативные способы отображения сущностей в таблицы базы данных

Иногда бывает полезно не отображать класс сущности в таблицу базы данных «как есть». Вместо связи между двумя классами, возможно, вы захотите объединить оба класса в одну таблицу. Такой подход позволяет загружать только часть таблицы при использовании одной из сущностей, что улучшит производительность запроса.

В этом разделе описаны пять альтернативных способов отображения классов в базу данных. У каждого из них есть преимущества в определенных ситуациях:

- *собственные типы* (*Owned types*) – позволяет объединить класс в таблицу класса сущности. Этот способ полезен для использования обычных классов для группировки данных;
- *таблица на иерархию* (*Table per hierarchy, TPH*) – позволяет сохранять набор унаследованных классов в одной таблице, например классы `Dog`, `Cat` и `Rabbit`, которые наследуются от класса `Animal`;
- *таблица на тип* (*Table per type, TPT*) – отображает каждый класс в отдельную таблицу. Этот подход работает как и TPH, за исключением того, что каждый класс отображается в отдельную таблицу;
- *разделение таблицы* (*Table splitting*) – позволяет отобразить несколько классов сущностей в одну таблицу. Этот способ полезен, когда некоторые столбцы в таблице читаются чаще, чем другие;
- *контейнер свойств* (*Property bag*) – позволяет создать класс сущности с помощью словаря, который дает возможность создать отображение при запуске. Контейнеры свойств также используют две другие функции: отображение одного и того же типа в несколько таблиц и использование индексатора в классах сущности.

8.9.1 Собственные типы: добавление обычного класса в класс сущности

В EF Core есть *собственные типы*, позволяющие определять класс, содержащий целую группу данных, таких как адрес или данные аудита,

которые вы, возможно, захотите использовать в нескольких местах своей базы данных. У класса собственного типа нет своего первичного ключа, поэтому у него нет своей идентичности; он полагается на уникальность сущности, которая «владеет» им. Говоря терминами предметно-ориентированного проектирования, собственные типы известны как *объекты-значения*.

> **EF6** Собственные типы EF Core аналогичны сложным типам EF6.x. Самое большое изменение заключается в том, что нужно специально настроить собственный тип, тогда как EF6.x считает любой класс без первичного ключа сложным типом (что может вызывать ошибки). Собственные типы EF Core имеют дополнительную функцию по сравнению с реализацией EF6.x: данные в собственном типе можно настроить для сохранения в отдельной скрытой таблице.

Вот два способа использования этих типов:

- данные собственного типа хранятся в той же таблице, в которую отображается класс сущности;
- данные собственного типа хранятся в отдельной от класса сущности таблице.

ДАННЫЕ СОБСТВЕННОГО ТИПА ХРАНЯТСЯ В ТОЙ ЖЕ ТАБЛИЦЕ, ЧТО И КЛАСС СУЩНОСТИ

В качестве примера собственного типа мы создадим класс сущности `OrderInfo`, которому требуются два адреса: `BillingAddress` и `DeliveryAddress`. Эти адреса представляются классом `Address`, показанным в следующем листинге. Можно пометить класс `Address` как собственный тип, добавив к нему атрибут `[Owned]`. У собственного типа нет первичного ключа, как показано в нижней части листинга.

Листинг 8.15 Собственный тип `Address`, за которым следует класс сущности `OrderInfo`

```
public class OrderInfo
{
    public int OrderInfoId { get; set; }
    public string OrderNumber { get; set; }

    public Address BillingAddress { get; set; }
    public Address DeliveryAddress { get; set; }
}

[Owned]
public class Address
{
    public string NumberAndStreet { get; set; }
    public string City { get; set; }
```

Класс сущности OrderInfo с первичным ключом и двумя адресами

Два разных класса Address. Данные каждого такого класса будут включены в таблицу, в которую отображается OrderInfo

Атрибут [Owned] сообщает EF Core, что это собственный тип

У собственного типа нет первичного ключа

```
    public string ZipPostCode { get; set; }
    [Required]
    [MaxLength(2)]
    public string CountryCodeIso2 { get; set; }
}
```

Поскольку мы добавили атрибут `[Owned]` в класс `Address` и поскольку мы используем собственный тип в той же таблице, нам не нужно использовать Fluent API для настройки собственного типа. Такой подход экономит время, особенно если собственный тип используется во многих местах, потому что нам не нужно писать конфигурацию Fluent API.

Но если вы не хотите использовать атрибут `[Owned]`, то в следующем листинге показано, как Fluent API сообщает EF Core, что свойства `BillingAddress` и `DeliveryAddress` в классе сущности `OrderInfo` – это собственные типы, а не связи.

Листинг 8.16 Fluent API для настройки собственных типов в `OrderInfo`

```
public class SplitOwnDbContext: DbContext
{
    public DbSet<OrderInfo> Orders { get; set; }
    //… Остальной код удален для ясности;

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBulder.Entity<OrderInfo>()
            .OwnsOne(p => p.BillingAddress);
        modelBulder.Entity<OrderInfo>()
            .OwnsOne(p => p.DeliveryAddress);
    }
}
```

Выбирает владельца собственного типа

Использует метод OwnsOne, чтобы сообщить EF Core, что свойство BillingAddress – это собственный тип и что данные нужно добавить в столбцы в таблице, в которую отображается OrderInfo

Повторяет процесс для второго свойства, DeliveryAddress

В результате у нас есть таблица, содержащая два скалярных свойства в классе сущности `OrderInfo`, за которым следуют два набора свойств класса `Address`, один с префиксом `BillingAddress_` и второй с префиксом `DeliveryAddress_`. Поскольку свойство собственного типа может иметь значение null, все свойства хранятся в базе данных в виде столбцов, допускающих значение null. Свойство `CountryCodeIso2` из листинга 8.15, например, помечено как `[Required]`, поэтому оно не должно иметь значения null, но, чтобы разрешить нулевое значение свойства для `BillingAddress` или `DeliveryAddress`, оно хранится в столбце, допускающем значение null. EF Core делает это, чтобы определить, нужно ли создавать экземпляр собственного типа при чтении сущности, содержащей собственный тип.

Тот факт, что свойство собственного типа может иметь значение null, означает, что собственные типы внутри класса сущности хорошо подходят для того, что в предметно-ориентированном проекти-

ровании называется *объект-значение*. У объекта-значения нет ключа, и два объекта-значения с одинаковыми свойствами считаются равными. Тот факт, что они могут иметь значение `null`, позволяет использовать «пустой» объект-значение.

> **ПРИМЕЧАНИЕ** Собственные типы, допускающие значение null, были введены в EF Core 3.0, но у них были проблемы с производительностью. (SQL использует `LEFT JOIN`.) В EF Core 5 эти проблемы были исправлены.

В следующем листинге показана часть команды SQL Server `CREATE TABLE`, которую EF Core создает для класса сущности `OrderInfo`, используя соглашение об именах.

Листинг 8.17 Команда SQL `CREATE TABLE`, показывающая имена столбцов

```
CREATE TABLE [Orders] (
    [OrderInfoId] int NOT NULL IDENTITY,
    [OrderNumber] nvarchar(max) NULL,
    [BillingAddress_City] nvarchar(max) NULL,
    [BillingAddress_NumberAndStreet] nvarchar(max) NULL,
    [BillingAddress_ZipPostCode] nvarchar(max) NULL,
    [BillingAddress_CountryCodeIso2] [nvarchar](2) NULL
    [DeliveryAddress_City] nvarchar(max) NULL,
    [DeliveryAddress_CountryCodeIso2] nvarchar(max) NULL,
    [DeliveryAddress_NumberAndStreet] nvarchar(max) NULL,
    [DeliveryAddress_CountryCodeIso2] [nvarchar](2) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([OrderInfoId])
);
```

У свойства есть атрибут [Required], но оно хранится как значение, допускающее null, для обработки нулевого адреса выставления счета/доставки

По умолчанию каждое свойство или поле в собственном типе хранится в столбце, допускающем значение null, даже если они не допускают этого значения. EF Core делает это, чтобы вы могли не создавать и не назначать экземпляр собственного типа. Всем столбцам, которые использует этот тип, устанавливается значение NULL. И если сущность с собственным типом считывается и все столбцы собственного типа имеют значение NULL, для свойства собственного типа устанавливается значение `null`.

Но в EF Core 5 была добавлена функция, позволяющая указать, что требуется собственный тип, то есть что он всегда должен присутствовать. Для этого в навигационное свойство `DeliveryAddress` `OrderInfo`, отображаемое в собственный тип (см. следующий листинг), добавляется метод Fluent API `IsRequired`. Кроме того, эта функция позволяет управлять допустимостью значений null для отдельных столбцов, чтобы следовать обычным правилам. Столбец `DeliveryAddress_CountryCodeIso2` из листинга 8.17, например, теперь NOT NULL.

Листинг 8.18 Fluent API для настройки собственных типов в `OrderInfo`

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
   modelBulder.Entity<OrderInfo>()
      .OwnsOne(p => p.BillingAddress);
   modelBulder.Entity<OrderInfo>()
      .OwnsOne(p => p.DeliveryAddress);

   modelBulder.Entity<OrderInfo>()
  .Navigation(p => p.DeliveryAddress)
       .IsRequired();
}
```

Выбирает навигационное свойство DeliveryAddress

Применение метода IsRequired означает, что свойство DeliveryAddress не должно быть нулевым

Использование собственных типов может помочь вам организовать вашу базу данных, превратив группы данных в эти типы. Это упрощает работу с группами данных, такими как `Address` и т. д., в вашем коде. Вот несколько заключительных моментов, касающихся собственных типов, хранящихся в классе сущности:

- навигационные свойства собственного типа, такие как `BillingAddress`, создаются автоматически и заполняются данными при чтении сущности. Метод `Include` или любая другая форма загрузки связей не требуется;
- Джули Лерман (@julielerman) отметила, что собственные типы могут заменить связи «один к нулю или к одному», особенно если собственный тип небольшой. Они обладают лучшей производительностью и загружаются автоматически. Это означает, что они были бы более подходящими реализациями `PriceOffer`, используемого в приложении Book App;
- собственные типы могут быть вложенными. Можно создать собственный тип `CustomerContact`, который, в свою очередь, содержит, например, собственный тип `Address`. Если вы используете тип `CustomerContact` в другом классе сущности – назовем его `SuperOrder`, – то все свойства `CustomerContact` и `Address` будут добавлены в таблицу `SuperOrder`.

ДАННЫЕ СОБСТВЕННОГО ТИПА РАЗМЕЩАЮТСЯ В ОТДЕЛЬНОЙ ОТ КЛАССА СУЩНОСТИ ТАБЛИЦЕ

Другой способ, которым EF Core может сохранять данные внутри собственного типа, – это отдельная таблица, а не таблица класса сущности. В этом примере мы создадим класс сущности `User`, у которого есть свойство `HomeAddress` типа `Address`. В этом случае мы добавляем вызов метода `ToTable` после метода `OwnsOne` в коде конфигурации.

Листинг 8.19 Настройка данных, которые будут храниться в отдельной таблице

```
public class SplitOwnDbContext: DbContext
{
    public DbSet<OrderInfo> Orders { get; set; }
    //… Остальной код удален для ясности;

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBulder.Entity<User>()
            .OwnsOne(p => p.HomeAddress);
            .ToTable("Addresses");
    }
}
```

Добавляя ToTable к OwnsOne, мы указываем EF Core хранить собственный тип Address в отдельной таблице с первичным ключом, равным первичному ключу сущности User, которая была сохранена в базе данных

EF Core устанавливает связь «один к одному», в которой первичный ключ также является внешним (см. раздел 8.6.1, вариант 3), а для состояния `OnDelete` установлено значение `Cascade`, поэтому запись о собственном типе основной сущности `User` удаляется. Следовательно, в базе данных есть две таблицы: Users и Addresses.

Листинг 8.20 Таблицы Users и Addresses в базе данных

```
CREATE TABLE [Users] (
    [UserId] int NOT NULL IDENTITY,
    [Name] nvarchar(max) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([UserId])
);
CREATE TABLE [Addresses] (
    [UserId] int NOT NULL IDENTITY,
    [City] nvarchar(max) NULL,
    [CountryCodeIso2] nvarchar(2) NOT NULL,
    [NumberAndStreet] nvarchar(max) NULL,
    [ZipPostCode] nvarchar(max) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([UserId]),
    CONSTRAINT "FK_Addresses_Users_UserId" FOREIGN KEY ("UserId")
        REFERENCES "Users" ("UserId") ON DELETE CASCADE
);
```

Обратите внимание, что свойства, не допускающие значения NULL, или свойства, допускающие это значение с параметром Required, теперь хранятся в столбцах, не допускающих значения NULL

Такое использование собственных типов отличается от первого варианта использования, при котором данные хранятся в таблице классов сущности, потому что вы можете сохранить экземпляр сущности `User` без адреса. Но те же правила применяются к запросам: свойство `HomeAddress` будет считано в запросе объекта `User` без необходимости использования метода `Include`.

Таблица Addresses, используемая для хранения данных `HomeAddress`, скрыта; к ней нельзя получить доступ через EF Core. Это может

быть хорошо или плохо в зависимости от ваших бизнес-потребностей. Но если вы хотите получить доступ к части `Address`, то можно реализовать ту же функциональность, используя два класса сущности со связью «один ко многим» между ними.

8.9.2 Таблица на иерархию (TPH): размещение унаследованных классов в одной таблице

При подходе «Таблица на иерархию» (TPH) все классы, наследуемые друг от друга, хранятся в одной таблице базы данных. Например, если вы хотите сохранить платеж в магазине, то этот платеж можно осуществить наличными (`PaymentCash`) или кредитной картой (`PaymentCard`). Каждый вариант содержит сумму (скажем, 10 долларов), но вариант с кредитной картой содержит дополнительную информацию, например квитанцию онлайн-транзакции. В этом случае TPH использует одну таблицу, чтобы сохранить все версии унаследованных классов и вернуть правильный тип сущности, `PaymentCash` или `PaymentCard` в зависимости от того, что было сохранено.

> **СОВЕТ** Я использовал классы TPH в паре проектов для своих клиентов и нахожу, что это хорошее решение для хранения схожих наборов данных, когда некоторые наборы нуждаются в дополнительных свойствах. Предположим, у вас было много типов продуктов с `Name`, `Price`, `ProductCode`, `Weight` и другими распространенными свойствами, но герметикам требуются свойства `MinTemp` и `MaxTemp`, которые TPH может реализовать, используя одну таблицу вместо множества.

TPH можно настроить «По соглашению», объединив все версии унаследованных классов в одну таблицу. У этого подхода есть преимущество, состоящее в том, что он сохраняет общие данные в одной таблице, но доступ к этим данным несколько затруднен, потому что у каждого унаследованного типа есть собственное свойство `DbSet<T>`. Но когда вы добавляете Fluent API, ко всем унаследованным классам можно получить доступ через одно свойство `DbSet<T>`, которое в нашем случае делает пример с `PaymentCash` и `PaymentCard` намного полезнее.

В первом примере используется несколько свойств `DbSet<T>`, по одному для каждого класса, и конфигурирование осуществляется «По соглашению». Во втором примере используется одно свойство `DbSet<T>`, отображенное в базовый класс. Я считаю, что это более полезная версия, и в ней показаны команды Fluent API для TPH.

НАСТРОЙКА TPH ПО СОГЛАШЕНИЮ

Чтобы применить подход «По соглашению» в примере с `PaymentCash` и `PaymentCard`, мы создаем класс `PaymentCash`, а затем класс `Payment-`

Card, который наследует от PaymentCash, как показано в следующем листинге. Видно, что PaymentCard наследует от PaymentCash и добавляет дополнительное свойство ReceiptCode.

Листинг 8.21 Два класса: PaymentCash и PaymentCard

```
public class PaymentCash
{
    [Key]
    public int PaymentId { get; set; }
    public decimal Amount { get; set; }
}
// PaymentCard - наследует от PaymentCash;
public class PaymentCard : PaymentCash
{
    public string ReceiptCode { get; set; }
}
```

В листинге 8.22, где используется подход «По соглашению», показаны возможности DbContext приложения с двумя свойствами DbSet<T>, по одному для каждого из двух классов. Поскольку мы включили оба класса, а PaymentCard наследует от PaymentCash, EF Core будет хранить оба класса в одной таблице.

Листинг 8.22 DbContext обновленного приложения с двумя свойствами DbSet<T>

```
public class Chapter08DbContext : DbContext
{
    //… Другие свойства DbSet<T> удалены;
   //Таблица на одну иерархию;
   public DbSet<PaymentCash> CashPayments { get; set; }
   public DbSet<PaymentCard> CreditPayments { get; set; }

   public Chapter08DbContext(
       DbContextOptions<Chapter08DbContext> options)
       : base(options)
   { }

   protected override void OnModelCreating
       (ModelBuilder modelBuilder)
   {
       //Никакой дополнительной настройки для PaymentCash или PaymentCard
         не требуется;
   }
}
```

Наконец, в этом листинге показан код, который EF Core сгенерирует для создания таблицы, в которой будут храниться классы сущностей PaymentCash и PaymentCard.

Листинг 8.23 SQL, сгенерированный EF Core для создания таблицы CashPayment

```
CREATE TABLE [CashPayments] (
    [PaymentId] int NOT NULL IDENTITY,
    [Amount] decimal(18, 2) NOT NULL,
    [Discriminator] nvarchar(max) NOT NULL,
    [ReceiptCode] nvarchar(max),
    CONSTRAINT [PK_CashPayments]
            PRIMARY KEY ([PaymentId])
);
```

Столбец Discriminator содержит имя класса, которое EF Core использует для определения типа сохраняемых данных. Если используется подход «По соглашению», этот столбец содержит имя класса в виде строки

Столбец ReceiptCode используется только в том случае, если это PaymentCredit

Как видите, EF Core добавил столбец Discriminator, используемый при получении данных из БД для создания правильного типа класса: `PaymentCash` или `PaymentCard` в зависимости от того, какие данные были сохранены. Кроме того, столбец ReceiptCode заполняется/читается только в том случае, если тип класса – это `PaymentCard`.

Любые скалярные свойства, не входящие в базовый класс TPH, отображаются в столбцы, допускающие значение null, потому что эти свойства используются только одной версией классов TPH. Если у вас много классов в классах TPH, стоит проверить, можно ли объединить похожие по типу свойства в один столбец. Например, в классах `Product` у вас может быть тип `Product "Sealant"`, для которого требуется вещественное свойство `MaxTemp`, и еще один тип `"Ballast"`, которому необходимо вещественное свойство `WeightKgs`. Можно отобразить оба свойства в один и тот же столбец, используя этот фрагмент кода:

```
public class Chapter08DbContext : DbContext
{
     //… Остальная часть не указана;

     Protected override void OnModelCreating
          (ModelBuilder modelBuilder)
     {
         modelBuilder.Entity<Sealant>()
              .Property(b => b.MaxTemp)
              .HasColumnName("DoubleValueCol");

         modelBuilder.Entity<Ballast>()
             .Property(b => b.WeightKgs)
             .HasColumnName("DoubleValueCol");
     }
}
```

ИСПОЛЬЗОВАНИЕ FLUENT API ДЛЯ УЛУЧШЕНИЯ ПРИМЕРА С TPH

Хотя подход «По соглашению» уменьшает количество таблиц в базе данных, у нас есть два отдельных свойства `DbSet<T>`, и вам нужно использовать какое-то одно из них, чтобы найти примененный платеж. Кроме того, у вас нет общего класса `Payment`, который можно было

бы использовать в любых других классах сущности. Но, немного по-другому скомпоновав код и добавив пару методов Fluent API, можно сделать это решение намного более полезным.

На рис. 8.12 показана новая компоновка. Вы создаете общий базовый класс, имея абстрактный класс `Payment`, от которого наследуются `PaymentCash` и `PaymentCard`. Этот подход позволяет использовать класс `Payment` в другом классе сущности `SoldIt`.

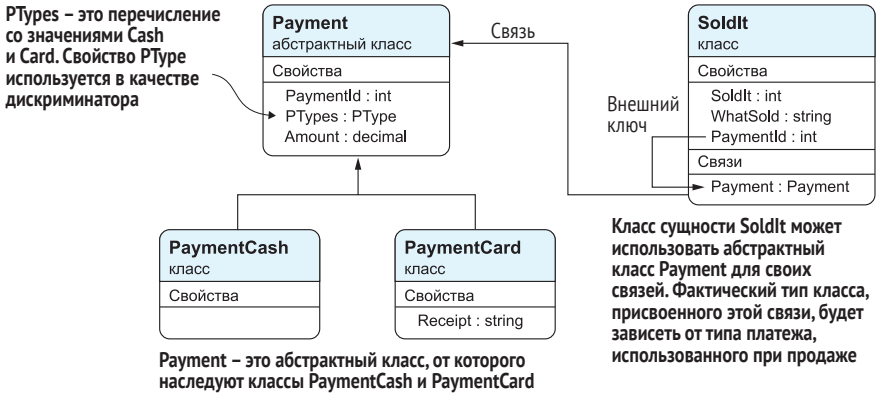


Рис. 8.12 Используя Fluent API, можно создать более удобную форму TPH. Здесь абстрактный класс `Payment` используется в качестве базового, и его можно применять внутри другого класса сущности. Фактический тип класса, помещенный в свойство платежа `SoldIt`, будет либо `PaymentCash`, либо `PaymentCard`, в зависимости от того, что использовалось при создании класса `SoldIt`

Этот подход намного полезнее, потому что теперь можно разместить абстрактный класс `Payment` в классе сущности `SoldIt` и получить сумму и тип платежа независимо от того, что это: оплата наличными или картой. Свойство `PType` сообщает вам тип (свойство `PType` имеет тип `PTypes`, представляющий собой перечисление со значениями `Cash` или `Card`), а если вам нужно свойство `Receipt` в `PaymentCard`, то можно привести класс `Payment` к типу `PaymentCard`.

Помимо создания классов сущностей, показанных на рис. 8.12, необходимо изменить DbContext и добавить конфигурацию Fluent API, чтобы сообщить EF Core о ваших классах TPH, поскольку они больше не соответствуют подходу «По соглашению». В этом листинге показан DbContext с конфигурацией столбца Discriminator.

Листинг 8.24 Измененный DbContext приложения с добавленными методами Fluent API

```
public class Chapter08DbContext : DbContext
{
    //… Другие свойства DbSet<T> удалены;
    public DbSet<Payment> Payments { get; set; }
```

Определяет свойство, через которое можно получить доступ ко всем платежам, как PaymentCash, так и PaymentCard

```
    public DbSet<SoldIt> SoldThings { get; set; }

    public Chapter08DbContext(
        DbContextOptions<Chapter08DbContext> options)
        : base(options)
    { }

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        //… Другие конфигурации удалены;
        modelBuilder.Entity<Payment>()
           .HasDiscriminator(b => b.PType)
           .HasValue<PaymentCash>(PTypes.Cash)
           .HasValue<PaymentCard>(PTypes.Card);
    }
}
```

Список проданных товаров, с обязательной ссылкой на Payment

Метод HasDiscriminator идентифицирует сущность как TPH, а затем выбирает свойство PType в качестве дискриминатора для различных типов. В данном случае это перечисление, размер которого установлен типом byte

Устанавливает значение дискриминатора для типа PaymentCash

ПРИМЕЧАНИЕ В этом примере в качестве базового класса используется абстрактный класс. Мне кажется, так будет полезнее, но с таким же успехом можно сохранить исходный `PaymentCash`, от которого наследует `PaymentCard`. Абстрактный базовый класс упрощает изменение общих свойств TPH.

ДОСТУП К СУЩНОСТЯМ С TPH

Теперь, когда мы настроили набор классов с TPH, рассмотрим различия в операциях CRUD. Большинство команд доступа к базе данных EF одинаковы, кроме некоторых различий в TPH-блоках. EF Core (как и EF6.x) отлично справляется с обработкой TPH.

Во-первых, создавать такие сущности несложно. Вы создаете экземпляр конкретного типа, который вам нужен. Следующий фрагмент кода создает сущность типа `PaymentCash` для продажи:

```
var sold = new SoldIt()
{
    WhatSold = "A hat",
    Payment = new PaymentCash {Amount = 12}
};
context.Add(sold);
context.SaveChanges();
```

Затем EF Core сохраняет правильную версию данных для этого типа и устанавливает дискриминатор, чтобы он знал тип класса экземпляра TPH. При чтении сущности `SoldIt`, которую вы только что сохранили, с методом `Include` для загрузки навигационного свойства `Payment` тип загруженного экземпляра `Payment` будет именно таким (`PaymentCash` или `PaymentCard`), какой использовался, при записи в базу данных. Кроме того, в этом примере свойство `Payment PType`, которое вы устанавливаете в качестве дискриминатора, сообщает тип оплаты: `Cash` или `Card`.

Когда вы запрашиваете данные TPH, метод EF Core `OfType<T>` позволяет фильтровать данные, чтобы найти экземпляры определенного класса. Запрос `context.Payments.OfType<PaymentCard>()` вернет, например, только платежи с использованием карты. Также можно отфильтровать классы TPH в методах `Include`. Смотрите эту статью для получения дополнительной информации: http://mng.bz/QmBj.

8.9.3 Таблица на тип (TPT): у каждого класса своя таблица

В EF Core версии 5 добавлена опция «таблица на тип» (TPT), позволяющая каждому классу сущности, который наследуется от базового класса, иметь собственную таблицу. Этот вариант представляет собой противоположность подходу «таблица на иерархию» (TPH), описанному в разделе 8.9.2. TPT – хорошее решение, если каждый класс в унаследованной иерархии содержит много разной информации; TPH лучше подходит, когда у каждого унаследованного класса большая общая часть и только небольшое количество различных данных по классам.

В качестве примера мы создадим TPT-решение для двух типов контейнеров: отгрузочных, которые используются на сухогрузных судах, и пластиковых, такие как бутылки, банки и коробки. У обоих типов контейнеров общие свойства высоты, длины и глубины, а остальные разные. В следующем листинге показаны три класса сущности: базовый абстрактный класс `Container` и классы `ShippingContainer` и `PlasticContainer`.

Листинг 8.25 Три класса, используемые в примере с TPT

```
public abstract class Container
{
    [Key]
    public int ContainerId { get; set; }

    public int HeightMm { get; set; }
    public int WidthMm { get; set; }
    public int DepthMm { get; set; }
}

public class ShippingContainer : Container
{
    public int ThicknessMm { get; set; }
    public string DoorType { get; set; }
    public int StackingMax { get; set; }
    public bool Refrigerated { get; set; }
}

public class PlasticContainer : Container
{
```

Класс Container помечен как абстрактный, потому что он не будет создаваться

Становится первичным ключом для каждой таблицы TPT

Общая часть каждого контейнера – это высота, ширина и глубина

Класс наследует от класса Container

Эти свойства уникальны для отгрузочного контейнера

```
    public int CapacityMl { get; set; }
    public Shapes Shape { get; set; }
    public string ColorARGB { get; set; }
}
```

Эти свойства уникальны для пластикового контейнера

Затем необходимо настроить DbContext приложения. Здесь два этапа: (a) добавить свойство `DbSet<Container>`, которое будет использоваться для доступа ко всем контейнерам, и (b) установить другие типы контейнеров, `ShippingContainer` и `PlasticContainer`, для отображения в их собственные таблицы. Эти две части показаны в следующем листинге.

Листинг 8.26 Обновления DbContext приложения для настройки контейнеров TPT

```
public class Chapter08DbContext : DbContext
{
    public Chapter08DbContext(
        DbContextOptions<Chapter08DbContext> options)
        : base(options)
{ }
//… Другие свойства DbSet<T> удалены для ясности;
public DbSet<Container> Containers { get; set; }

protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    //… Другие конфигурации удалены для ясности;

    modelBuilder.Entity<ShippingContainer>()
        .ToTable(nameof(ShippingContainer));
    modelBuilder.Entity<PlasticContainer>()
        .ToTable(nameof(PlasticContainer));
}
```

Это единственное свойство DbSet используется для доступа к различным контейнерам

Эти методы Fluent API отображают каждый контейнер в отдельную таблицу

Результатом обновления DbContext приложения являются три таблицы:

- таблица `Containers` через DbSet, содержащая общие данные для каждой записи;
- таблица `ShippingContainer`, содержащая свойства `Container` и `ShippingContainer`;
- таблица `PlasticContainer`, содержащая свойства `Container` и `PlasticContainer`.

Таблицы `ShippingContainer` и `PlasticContainer` добавляются обычным способом: используя метод `context.Add`. Но магия наступает, когда вы запрашиваете `DbSet<Container> Containers` в DbContext, потому что он возвращает все контейнеры, используя правильный тип класса, `ShippingContainer` или `PlasticContainer`, для каждой возвращаемой сущности.

Есть несколько вариантов загрузки одного типа классов TPT. Вот три подхода. Наиболее эффективный приведен в конце:

- *прочитать весь запрос* – `context.Containers.ToList()`.
  Здесь вы читаете все типы TPT, и каждая запись в списке будет относиться к правильному типу (`ShippingContainer` или `PlasticContainer`) для возвращаемого типа. Этот подход полезен только в том случае, если вы хотите перечислить все контейнеры;
- запрос `OfType` – `context.Containers.OfType<ShippingContainer>().ToList()`.
  Здесь считываете только записи типа `ShippingContainer`;
- запрос `Set` – `context.Set<ShippingContainer>().ToList()`.
  Здесь вы возвращаете только тип `ShippingContainer` (как и при запросе `OfType`), но SQL несколько эффективнее, чем запрос `OfType`.

8.9.4 Разбиение таблицы: отображение нескольких классов сущностей в одну и ту же таблицу

Следующая функция, *разбиение таблицы*, позволяет отображать несколько сущностей в одну и ту же таблицу. Это полезная функция, если у вас большой объем данных для хранения для одной сущности, но для обычных запросов к этой сущности требуется всего несколько столбцов. Разбиение таблицы похоже на встраивание запроса `Select` в класс сущности; запрос будет быстрее, потому что вы загружаете только подмножество данных всей сущности. Кроме того, разделение таблицы на два или более классов ускоряет обновление.

В этом примере есть два класса сущности: `BookSummary` и `BookDetail`, – которые оба отображаются в таблицу базы данных Books. На рис. 8.13 показан результат настройки этих двух классов сущностей в виде разбиения таблицы.

Вот код конфигурации:

Листинг 8.27 Настройка разбиения таблицы для классов `BookSummary` и `BookDetail`

```
public class SplitOwnDbContext : DbContext
{
    //… Остальной код удален;

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BookSummary>()
           .HasOne(e => e.Details)
           .WithOne()
           .HasForeignKey<BookDetail>
                (e => e.BookDetailId);
```

Определяет, что у двух книг та же связь, как если бы вы установили связь «один к одному»

В этом случае метод HasForeignKey должен ссылаться на первичный ключ в сущности BookDetail

```
        modelBuilder.Entity<BookSummary>()
            .ToTable("Books");

        modelBuilder.Entity<BookDetail>()
            .ToTable("Books");
    }
}
```

Нужно отобразить оба класса сущности в таблицу Books, чтобы инициировать разбиение таблицы

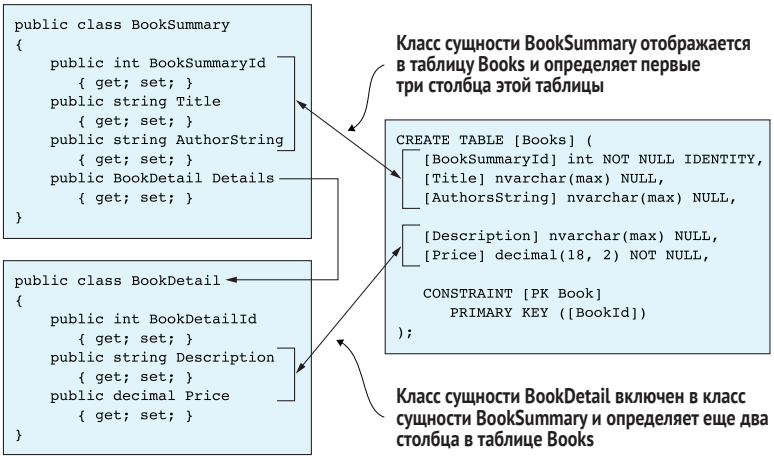


Рис. 8.13 Результат использования функции разбиения таблицы в EF Core для отображения двух классов сущностей `BookSummary` и `BookDetail` в одну таблицу Books. Книге нужно много информации, но для большинства запросов требуется только часть `BookSummary`. В результате создается предварительно отобранный набор столбцов для более быстрого запроса

После того как вы настроили две сущности через разбиение таблицы, можно запросить сущность `BookSummary` и получить сводные данные. Чтобы получить часть `BookDetails`, можно либо запросить сущность `BookSummary` и загрузить свойство связи `Details` (скажем, с помощью метода `Include`), либо прочитать только часть `BookDetails` прямо из базы данных.

ПРИМЕЧАНИЕ В третьей части книги я создам гораздо более сложную версию приложения Book App, используя реальные данные из Manning Publications. Я использую разбиение таблицы для разделения больших описаний, используемых в подробном отзыве на книгу, от основной части данных Book. Любые обновления, скажем свойство `Book`, `PublishedOn`, намного быстрее, потому что мне не нужно читать все описания.

Прежде чем закончить с этой темой, позвольте мне отметить несколько моментов:

- можно обновить отдельный класс сущности в разбиении таблицы; не нужно загружать все сущности, участвующие в разбиении таблицы, чтобы выполнить обновление;

- вы видели, что таблица разделена на два класса сущности, но количество классов может быть любым;
- если у вас есть маркеры параллелизма (см. раздел 10.6.2), они должны быть во всех классах сущности, отображаемых в одну и ту же таблицу, чтобы убедиться, что значения маркера не выходят за рамки данных, когда обновляется только один из классов сущностей, отображаемый в таблицу.

8.9.5 Контейнер свойств: использование словаря в качестве класса сущности

В EF Core 5 была добавлена функция, *контейнер свойств*, в которой используется тип `Dictionary<string, object>` для отображения в базу данных. Контейнер свойств используется для реализации прямой связи «многие ко многим», при которой связующая таблица должна создаваться во время конфигурации. Вы также можете использовать контейнер свойств, но он полезен только в определенных областях, таких как создание сущности «контейнер свойств» в таблице, структура которой определяется внешними данными.

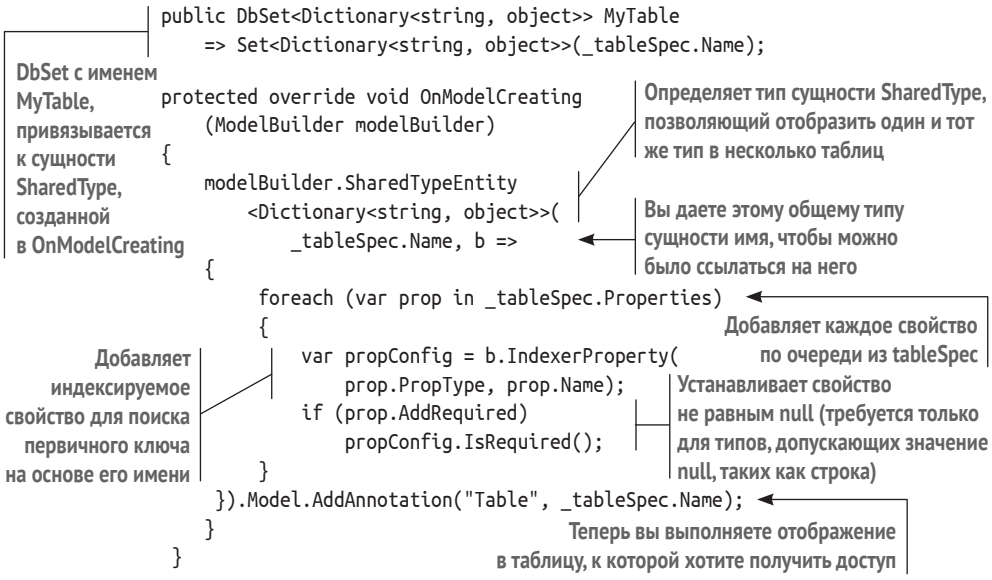
> **ПРИМЕЧАНИЕ** Контейнер свойств использует две функции, которые не описаны в других разделах книги. Первая – это *общие типы сущностей*, где один и тот же тип можно отображать в несколько таблиц. Вторая функция использует свойство индексатора C# в классе сущности для доступа к данным, например `public object this[string key]` ….

Допустим, вы отображаете контейнер свойств в таблицу, имя и столбцы которой определяются внешними данными, а не структурой класса. В этом примере таблица определена в классе `TableSpec`, который, как предполагается, был прочитан при запуске, возможно, из файла appsettings.json. В следующем листинге показан DbContext приложения с необходимым кодом для настройки и доступа к таблице через контейнер свойств.

Листинг 8.28 Использование контейнера свойств `Dictionary` для определения таблицы при запуске

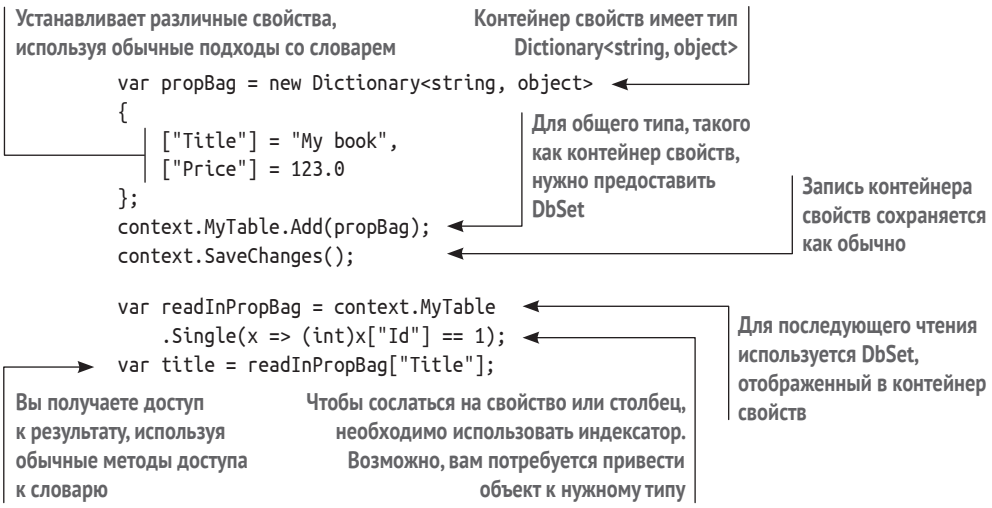
```
public class PropertyBagsDbContext : DbContext
{
    private readonly TableSpec _tableSpec;

    public PropertyBagsDbContext(
        DbContextOptions<PropertyBagsDbContext> options,
        TableSpec tableSpec)
        : base(options)
    {
        _tableSpec = tableSpec;
    }
```

Вы передаете класс, содержащий спецификацию таблицы и свойств



Для ясности: данные в классе `TableSpec` должны каждый раз быть одинаковыми, потому что EF Core кеширует конфигурацию. Конфигурация контейнера свойств является фиксированной на все время работы приложения. Чтобы получить доступ к этой сущности, используется свойство `MyTable`, показанное в следующем листинге. В этом листинге показано добавление новой записи через словарь, а затем ее чтение, включая доступ к свойствам контейнера свойств в запросе LINQ.

Листинг 8.29 Добавление и запрос контейнера свойств



Этот листинг – конкретный пример, в котором контейнер свойств – хорошее решение, но его можно настроить вручную. Вот еще некоторые сведения:

- Имена свойств контейнера свойств соответствуют именованию «По соглашению». Например, первичный ключ – это `"Id"`. Но можно переопределить этот параметр с помощью команд Fluent API, как обычно.
- Контейнеров свойств может быть несколько. Метод Fluent API `SharedTypeEntity` позволяет отображать один и тот же тип в разные таблицы.
- Контейнер свойств может иметь связи с другими классами или контейнерами свойств. Вы используете методы Fluent API `HasOne`/`HasMany`, но не можете определять навигационные свойства в контейнере свойств.
- Не нужно указывать каждое свойство в словаре, когда вы добавляете контейнер свойств. Для всех незаданных свойств или столбцов будет установлено значение типа по умолчанию.

Резюме

- Если вы следуете правилам именования «По соглашению» для внешних ключей, EF Core может найти и настроить самые обычные связи.
- Две аннотации данных позволяют решить несколько конкретных проблем, связанных с внешними ключами, с именами, которые не соответствуют правилам именования «По соглашению».
- Fluent API – это наиболее полный способ настройки связей. Некоторые функции, такие как установка действия при удалении зависимой сущности, доступны только в Fluent API.
- Можно автоматизировать какую-то конфигурацию классов сущностей, добавив код, который выполняется в методе DbContext `OnModelCreating`.
- EF Core позволяет управлять обновлениями навигационных свойств, включая остановку, добавление или удаление записей в навигационных свойствах коллекции.
- EF Core предоставляет множество способов отображения классов сущностей в таблицу базы данных. Основные способы – это собственные типы, таблица на иерархию, таблица на тип, разбиение таблицы и контейнеры свойств.

Для читателей, знакомых с EF6:

- базовый процесс настройки связей в EF Core такой же, как и в EF6.x, но команды Fluent API значительно изменились;

- EF6.x добавляет внешние ключи, если вы забыли добавить их самостоятельно, но они недоступны с помощью обычных команд EF6.x. EF Core позволяет получить к ним доступ через теневые свойства;
- в EF Core 5 добавлена функция, аналогичная связи «многие ко многим» в EF6.x. Теперь EF Core автоматически создает связующую таблицу (см. раздел 3.4.4), но реализация EF Core отличается от того, как EF6.x реализует эту функцию;
- EF Core представила такие нововведения, как доступ к теневым свойствам, альтернативные ключи и резервные поля;
- собственные типы EF Core предоставляют функции, которые можно было бы найти в сложных типах EF6.x. Дополнительные функции включают в себя хранение собственных типов в их собственной таблице;
- TPH, TPT и разбиения таблиц в EF Core аналогичны соответствующим функциям в EF6.x, но собственные типы и контейнеры свойств отсутствуют в EF6.x.

9 Управление миграциями базы данных

В этой главе рассматриваются следующие темы:

- различные способы создания команд для обновления структуры базы данных;
- три отправные точки, используя которые, вы создаете изменения в структуре базы данных;
- как обнаружить и исправить изменения в структуре базы данных, которые могут привести к потере данных;
- как характеристики приложения влияют на способ применения изменения к базе данных.

В этой главе рассматриваются способы изменения структуры базы данных, именуемые миграцией базы данных. Структура базы данных называется *схемой базы данных*; она состоит из таблиц, столбцов, ограничений и т. д., составляющих базу данных. Создание и обновление схемы базы данных может показаться простым делом, потому что EF Core предоставляет метод `Migrate`, который сделает все за вас: вы создаете классы сущностей и добавляете немного конфигурации, а EF Core создает красивую блестящую базу данных.

Проблема состоит в том, что данный метод скрывает целую серию проблем, связанных с миграцией, которые не сразу очевидны. Например, переименование свойства в классе сущности по умолчанию вызывает удаление столбца базы данных этого свойства вместе со всеми данными, которые там были! Поэтому в этой главе, помимо подробного описания того, как создавать и применять миграции, я освещаю

ключевые вопросы, которые нужно учитывать при обновлении базы данных. Никому не хочется вывести из строя реальную базу данных.

Есть превосходная документация по миграциям в EF Core (http://mng.bz/XdR6), поэтому в этой главе я не пытаюсь дублировать ее. Наоборот. В ней рассматриваются нюансы и проблемы, связанные с миграцией базы данных, а также их плюсы и минусы. Есть много способов создать и применить миграцию, и в этой главе описаны разные варианты. Кроме того, приводятся примеры, как справиться с более сложными проблемами, такими как правильная обработка миграций, которые могут привести к потере данных, и применение миграции к базе данных во время работы приложения. Эти знания помогут выбрать правильный подход к созданию миграции и успешно применить ее к базе данных.

9.1 Как устроена эта глава

Эта глава начинается с раздела 9.2, знакомящего вас с базами данных, которым необходима миграция, и важной проблемой, касающейся гарантии сохранности данных во время миграции. После этого раздела идут две части, посвященные созданию и применению миграций:

- часть 1, создание миграции базы данных, начинается с раздела 9.3. В ней изложены три подхода к созданию миграции или созданию классов EF Core и конфигурации для соответствия уже существующей базе данных;
- часть 2, применение миграции к базе данных, начинается с раздела 9.8. Здесь описываются способы применения миграции к рабочей базе данных, включая сложности ее обновления во время работы приложения.

Эти части охватывают множество подходов, которые нужно рассмотреть. В каждой части есть таблица, в которой перечислены плюсы, минусы и ограничения каждого из подходов, которые должны помочь вам сделать правильный выбор для вашего проекта.

9.2 Сложности изменения базы данных приложения

В этом разделе рассказывается о проблемах, связанных с миграцией базы данных, особенно той базы данных, которую использует ваше приложение. Освещаемые здесь темы являются общими для всех реляционных баз данных и любой программной системы. Есть множество способов организовать развертывание базы данных и приложения, и у каждого из них свои компромиссы по сложности, масштабируемости, доступности и трудозатратам на разработку и эксплуатацию (DevOps).

Объединив сведения из этой главы с тем, что вам известно о вашем приложении, вы сможете решить, какой подход использовать для создания и миграции базы данных. Наличие продуманного плана или политики создания и применения миграций сделает этот процесс более безопасным и быстрым.

9.2.1 Какие базы данных нуждаются в обновлении

Прежде чем перейти к описанию обновления схемы базы данных, рассмотрим базы данных, которые могут быть использованы в разрабатываемом приложении. На рис. 9.1 показана возможная организация команды разработчиков, состоящей из нескольких человек, и этапы работы: разработка, тестирование, обкатка и промышленная эксплуатация.

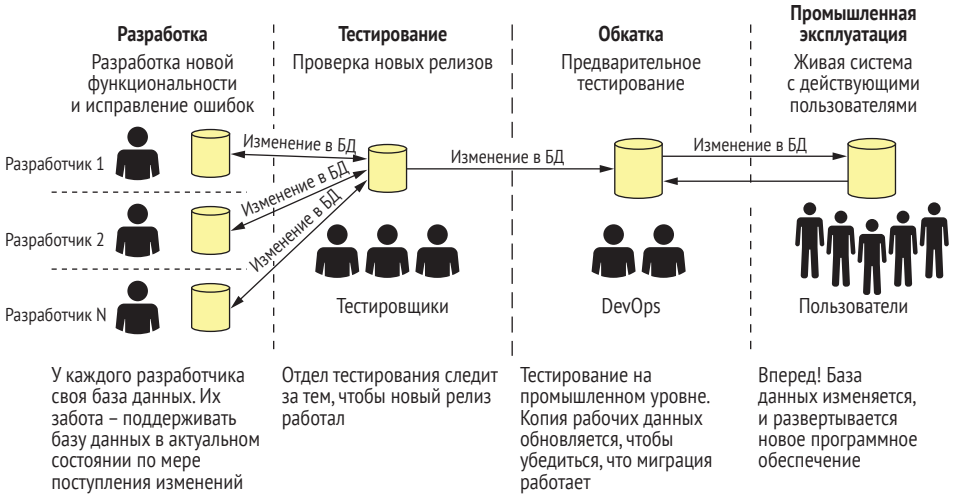


Рис. 9.1 При разработке приложения могут использоваться различные базы данных, и все они потребуют внесения изменений в схему базы данных. Термины «разработка», «тестирование», «обкатка» и «промышленная эксплуатация» относятся к различным частям разработки, тестирования и развертывания приложения и любым связанным изменениям схемы базы данных

Не во всех проектах по разработке есть все эти этапы, а у некоторых их больше или они отличаются. Также этот рисунок предполагает, что в промышленном окружении используется только одна база данных, но у вас может быть несколько копий одной и той же базы данных. Кроме того, у вас могут быть разработчики, совместно использующие одну базу данных, но у такого подхода есть ограничения; смотрите примечание ниже. Комбинаций может быть бесконечно много. В этой главе речь идет о базах данных, используемых в окружении разработки и промышленном окружении, но имейте в виду, что обновления схемы базы данных могут быть необходимы и в других базах.

ПРИМЕЧАНИЕ Использование одной общей базы данных в окружении разработки может сработать, но здесь есть ограничения. Разработчики могут, например, применить миграцию к базе данных, прежде чем объединят код в основную ветку, что может вызвать проблемы. Раздел 9.2.2 познакомит вас с миграциями, которые могут вызывать проблемы.

9.2.2 Миграция, которая может привести к потере данных

Полезно разделить миграции на две группы: некритическое изменение и критическое изменение, нарушающее целостность данных. *Некритическое изменение* – это изменение, при котором таблицы или столбцы, в которых есть полезные данные, не удаляются, а в ходе *критического изменения, приводящего к потере данных*, эти таблицы или столбцы удаляются. Поэтому если вы не хотите потерять важные данные, нужно добавить дополнительный этап копирования к миграции второго типа, чтобы данные были сохранены.

К счастью, в разрабатываемых приложениях многие миграции относятся к первому типу, потому что вы добавляете новые таблицы и столбцы в свою базу данных. Но иногда нужно реструктурировать базу данных таким образом, что нужно переместить столбцы из одной таблицы в другую, возможно в новую, таблицу. В разделе 9.5 приведены два примера изменений, приводящих к потере данных, и рассказывается, как их исправить:

- переименование свойства (раздел 9.5.1);
- перемещение столбцов из одной таблицы в другую (раздел 9.5.2).

ПРИМЕЧАНИЕ В разделе 9.8 обсуждается еще один тип критических изменений: критическое изменение приложения. Оно относится к миграции, которая может вызвать ошибки в работающем приложении. Это изменение имеет значение, если вы пытаетесь выполнить миграцию во время работы приложения.

9.3 Часть 1: знакомство с тремя подходами к созданию миграции

Раздел 9.2 применим к любой форме миграции базы данных, но теперь основное внимание будет уделено EF Core. Это важно, потому что задача заключается не только в изменении базы данных; это необходимо и для того, чтобы измененная база данных соответствовала классам сущностей и конфигурации EF Core, хранящейся в DbContext приложения. Если вы используете инструменты миграции EF Core, то предполагается, что база данных будет соответствовать DbContext, но, как вы увидите, во многих других подходах к миграции базы данных это совпадение не гарантируется.

Есть три основных способа создать обновленную базу данных, соответствующую DbContext приложения. У каждого подхода есть своя отправная точка, которую Артур Викерс (технический менеджер из команды EF Core) называет *источником истины*:

- *использование миграций с EF Core* – данный подход учитывает классы сущностей, а конфигурация EF Core – это источник истины. Такой подход к миграциям – самый простой, но сложные проблемы, такие как борьба с критическими изменениями, приводящими к потере данных, требуют, чтобы миграции корректировались вручную;
- *использование сценариев SQL для создания миграций* – при таком подходе источник истины – это команды SQL, используемые для создания и миграции базы данных. У вас есть полный контроль над схемой базы данных, и сюда можно включать функции, настройкой которых EF Core не занимается, например ограничения на уровне столбца. Но и здесь есть серьезная проблема – согласование своих изменений в SQL с внутренней моделью EF Core;
- *использование инструмента обратного проектирования EF Core* – при таком подходе источником истины является база данных. Вы повторно создаете классы сущностей и DbContext приложения со всеми необходимыми конфигурациями. Этот подход используется в основном для создания приложения EF Core вокруг существующей базы данных.

На рис. 9.2 представлен обзор пяти способов миграции базы данных и их ключевые особенности. Каждый раздел, посвященный миграции, начинается с таблицы с кратким описанием подхода и моими взглядами на то, когда эти подходы полезны.

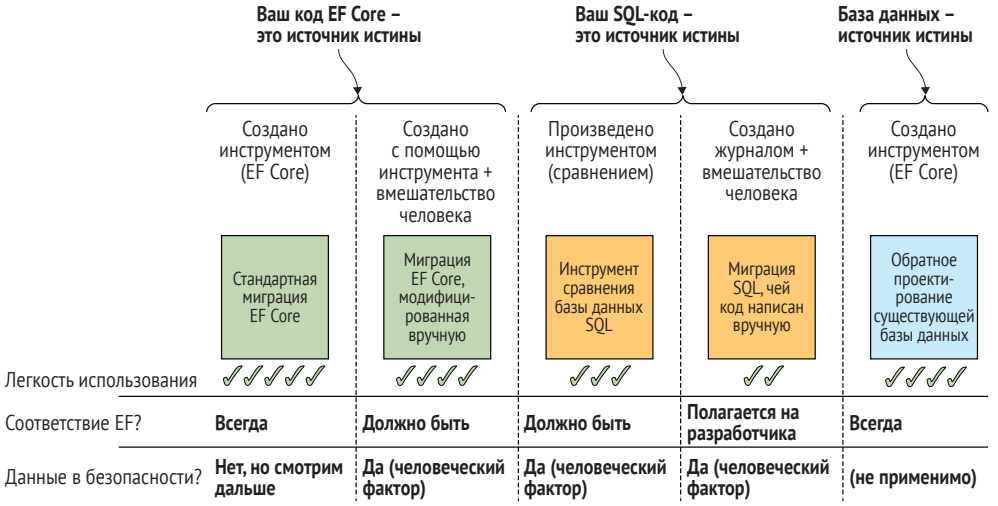


Рис. 9.2 Простой обзор пяти способов миграции базы данных и проверки базы данных на предмет ее соответствия внутренней модели EF Core

9.4 Создание миграции с помощью команды EF Core add migration

Инструменты миграции – это стандартный способ создания и обновления базы данных от EF Core. Это самый простой подход, потому что он автоматически создает корректные команды SQL для обновления, избавляя вас от необходимости погружаться во внутренности базы данных и язык SQL, чтобы создать или изменить базу данных приложения.

Мы начнем с изучения стандартной миграции, производимой инструментами EF Core без дополнительного редактирования с вашей стороны. Стандартная миграция может справиться с большинством ситуаций и формирует основу для изменения миграции, если это необходимо. Обычно миграцию нужно редактировать, когда вы имеете дело с такими вещами, как критические изменения, приводящие к потере данных (раздел 9.5), но сначала ознакомьтесь с возможностями стандартного варианта.

Стандартную миграцию можно создать с помощью инструментов EF Core, в частности команды `add migration`. Она использует классы сущностей и DbContext приложения, а источником истины является его конфигурационный код. Но этим командам также необходимо знать предыдущее состояние модели базы данных, чтобы решить, что нужно изменить. Для этого нужно посмотреть на класс, созданный при последнем запуске инструментов миграции, которые содержат снимок модели базы данных. Что касается первой миграции, то этого класса еще не будет, поэтому инструменты миграции предполагают, что у базы данных пустая схема – т. е. нет таблиц, индексов и т. д.

Поэтому когда вы выполняете команду `add migration`, она сравнивает снимок с вашими текущими классами сущностей, а DbContext приложения – с его кодом конфигурации. На основе этих данных можно определить, что изменилось; затем она создает два класса, содержащих команды для добавления изменений в базу данных (рис. 9.3).

Не создавайте классы сущностей так же, как обычные классы

EF Core отлично подходит для того, чтобы ваша база данных выглядела как обычные классы, но не нужно создавать классы сущностей точно так же, как и обычные классы. Например, в обычных классах хороший способ остановить дублирование – это использовать свойства, которые обращаются к другим свойствам, известным как *определения тела выражения*. Например:

```
public string FullName => $"{FirstName} {LastName}";
```

Этот метод подходит для обычного класса, но если вы используете его для класса сущности, то запрос, выполняющий фильтрацию или сортировку по свойству `FullName`, завершится ошибкой. В этом случае вам по-

надобится предоставить свойство, связанное со столбцом базы данных (возможно, используя новый, постоянный вычисляемый столбец; см. главу 10), чтобы убедиться, что EF Core может сортировать и фильтровать эти данные.

Кроме того, нужно тщательно продумать, какие свойства и связи вы добавляете в класс сущности. Рефакторинг обычного класса прост, но для рефакторинга класса сущности требуется миграция и, возможно, копирование данных.

Помните, что классы сущностей с их навигационными свойствами определяют структуру базы данных. Тот факт, что EF Core позволяет легко определять эти вещи, не означает, что вам не следует думать о структуре базы данных и ее производительности.

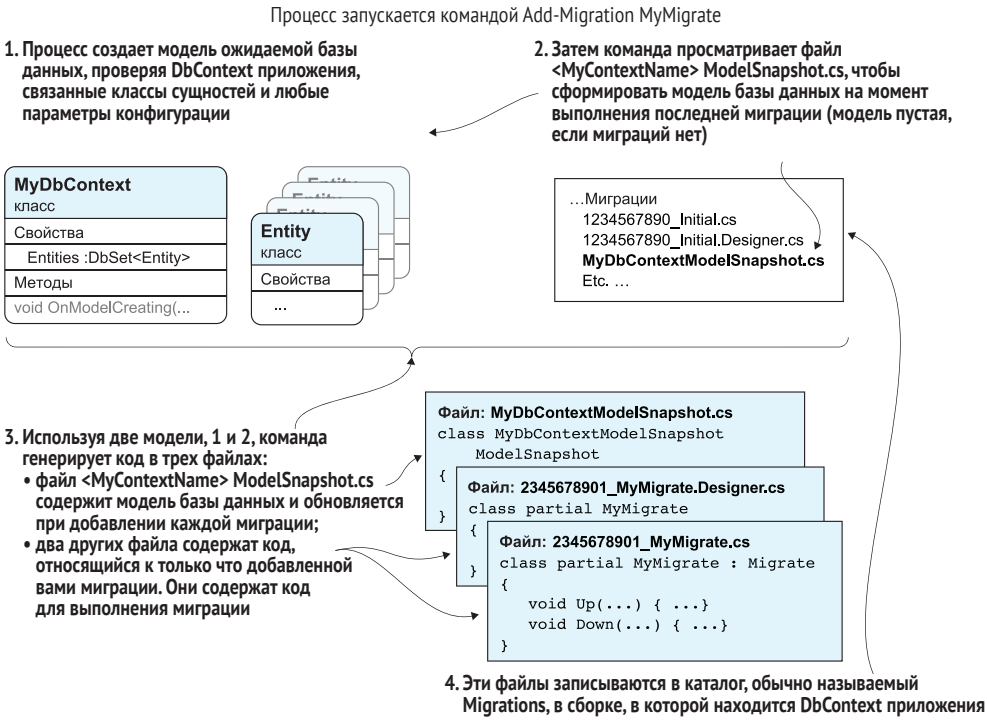


Рис 9.3 Выполнение команды `add migration` для создания новой миграции. Команда сравнивает две модели базы данных. Одна модель исходит из нашего текущего приложения с его DbContext, классами сущностей и конфигурацией EF Core; другая – из файла <MyContextName>ModelSnapshot.cs (который пуст, если это ваша первая миграция). Сравнивая эти две модели, EF Core может создать код, который обновит схему базы данных, чтобы соответствовать текущей модели базы данных EF Core

Прежде чем подробно изучить команду `add migration`, посмотрите на табл. 9.1. В ней приводится информация об использовании стандартной миграции для обновления схемы базы данных. В каждом

разделе, посвященном подходу к миграции, есть таблица, аналогичная этой, чтобы можно было сравнить функции и ограничения каждого подхода.

Таблица 9.1 Краткое изложение достоинств, недостатков и ограничений стандартной миграции, создаваемой с помощью команды `add migration`

	Примечания
Положительные стороны	▪ Автоматически создает правильную миграцию. ▪ Управляет заполнением базы данных. ▪ Не требует знания SQL. ▪ Включает функцию удаления миграции (см. раздел 9.4.4)
Отрицательные стороны	▪ Работает только в том случае, если ваш код является источником истины
Ограничения	▪ Стандартные миграции EF Core не могут обрабатывать критические изменения (но см. раздел 9.5). ▪ Стандартные миграции EF Core зависят от конкретной базы данных (но см. раздел 9.5.4)
Советы	Следите за сообщениями об ошибках при выполнении команды `add migration`. Если EF Core обнаруживает изменение, которое может привести к потере данных, то выводит сообщение об ошибке, но по-прежнему создает файлы миграции. *Нужно* изменять сценарий миграции в таких случаях, иначе вы потеряете данные (см.раздел 9.5.2)
Мой вердикт	Этот подход – простой способ работы с миграциями, и во многих случаях он хорошо работает. В первую очередь рассмотрите его, если код приложения управляет проектированием базы данных

СОВЕТ Я рекомендую видеоролик EF Core Community Standup, в котором рассказывается о некоторых функциях EF Core 5 и философии, лежащей в основе функций миграции EF Core, – http://mng.bz/yYmq.

9.4.1 Требования перед запуском любой команды миграции EF Core

Чтобы запустить любую команду инструментов миграции EF Core, необходимо установить требуемый код и настроить свое приложение определенным образом. Есть две версии инструментов миграции EF Core: инструменты командной строки (CLI) `dotnet-ef` и консоль диспетчера пакетов (PMC) Visual Studio.

Чтобы использовать инструменты командной строки, необходимо установить их на свой компьютер, используемый для разработки, через соответствующую команду. Следующая команда установит инструменты `dotnet-ef` глобально, дабы можно было использовать их в любом каталоге:

```
dotnet tool install --global dotnet-ef
```

Чтобы использовать PMC, необходимо включить пакет NuGet `Microsoft.EntityFrameworkCore.Tools` в основное приложение и пакет поставщика баз данных EF Core, например `Microsoft.EntityFrame-`

workCore.SqlServer, в проект, содержащий DbContext приложения, к которому вы хотите применить миграцию.

Эти инструменты должны иметь возможность создавать экземпляр DbContext. Если ваш проект представляет собой веб-узел ASP.NET Core или универсальный хост .NET Core, то инструменты могут использовать его для получения экземпляра DbContext, установленного в классе Startup.

Если вы не используете ASP.NET Core, то можно создать класс, реализующий интерфейс `IDesignTimeDbContextFactory<TContext>`. Этот класс должен находиться в том же проекте, что и DbContext, к которому вы хотите применить миграцию. В следующем листинге показан пример, взятый из ветки Part2 связанного репозитория GitHub.

Листинг 9.1 Класс, предоставляющий экземпляр DbContext инструментам миграции

Инструменты EF Core используют этот класс для получения экземпляра DbContext

С помощью данного интерфейса инструменты EF Core находят и создают этот класс

```
public class DesignTimeContextFactory
    : IDesignTimeDbContextFactory<EfCoreContext>
{
    private const string connectionString =
        "Server=(localdb)\\mssqllocaldb;Database=..."

    public EfCoreContext CreateDbContext(string[] args)
    {
        var optionsBuilder =
            new DbContextOptionsBuilder<EfCoreContext>();
        optionsBuilder.UseSqlServer(connectionString);

        return new EfCoreContext(optionsBuilder.Options);
    }
}
```

Необходимо предоставить строку подключения к вашей локальной базе данных

Интерфейсу требуется этот метод, который возвращает экземпляр DbContext

Вы используете обычные команды для настройки поставщика базы данных

Возвращает DbContext, который будет использоваться инструментами EF Core

9.4.2 Запуск команды add migration

Чтобы создать миграцию EF Core, необходимо выполнить команду `add migration` из командной строки (CLI инструменты) или в окне PMC. Два способа миграции базы данных, инструменты командной строки и PMC, имеют разные имена и параметры. В следующем списке показана команда `add migration`, которую я использовал для создания миграции в приложении Book App. Обратите внимание, что версия с командной строкой была запущена в каталоге проекта ASP.NET.Core BookApp:

- *CLI:* `dotnet ef migrations add Ch09Migrate -p ../DataLayer`;
- *PMC:* `Add-Migration Ch09Migrate -Project DataLayer`.

ПРИМЕЧАНИЕ Есть обширный набор команд со множеством параметров, и потребуется немало страниц, чтобы воспроизвести документацию по EF Core. Поэтому рекомендую вам обратиться к справочнику на странице http://mng.bz/MXEn.

9.4.3 Заполнение базы данных с помощью миграции

Миграции EF Core могут содержать данные, которые будут добавлены в базу данных. Этот процесс известен как заполнение базы данных. Неплохой пример использования этой функциональности – добавление констант в базу данных, например типов продуктов и клиентов для сайта онлайн-магазина. Должен сказать, что начальные данные можно изменить, поэтому они не являются постоянными. Но изменить их можно только с помощью миграции, поэтому лучше использовать ее для данных, которые не меняются (или меняются очень редко).

ПРИМЕЧАНИЕ Помимо добавления исходных данных при применении миграции, метод `context.Database.EnsureCreated()` (обычно используется в модульном тестировании) заполняет созданную базу данных. Смотрите главу 17 для получения дополнительной информации о модульном тестировании.

Исходные данные добавляются через конфигурацию Fluent API, используя метод `HasData`. В листинге 9.2 приводится пример того, как связать исходные данные через первичный и внешний ключи. Здесь содержатся исходные данные, которые сложнее тех, что обычно есть у меня, но я предоставляю их для демонстрации различных способов их настройки. Вот классы, используемые в этом примере:

- класс сущности `Project` с `ProjectManager` типа `User`;
- класс сущности `User`, который содержит имя и адрес пользователя;
- класс `Address`, собственный тип (см. раздел 8.9.1), содержащий адрес.

Листинг 9.2 Пример настройки исходных данных с помощью метода `HasData`

Заполнение настраивается через Fluent API

Добавляет два проекта по умолчанию. Обратите внимание, что вы должны предоставить первичный ключ

```
protected override void OnModelCreating( (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Project>().HasData(
        new { ProjectId = 1, ProjectName = "Project1"},
        new { ProjectId = 2, ProjectName = "Project2"});
    modelBuilder.Entity<User>().HasData(
        new { UserId = 1, Name = "Jill", ProjectId = 1 },
        new { UserId = 2, Name = "Jack", ProjectId = 2 });
```

Установка ProjectManager для каждого Project. Обратите внимание, что вы устанавливаете внешний ключ проекта, в котором они находятся

Класс User имеет собственный тип, в котором хранится адрес пользователя

Указываем адреса пользователей. Обратите внимание, что используется UserId, чтобы определить, для какого пользователя вы добавляете данные

```
    modelBuilder.Entity<User>()
        .OwnsOne(x => x.Address).HasData(
        new {UserId = 1, Street = "Street1", City = "city1"},
        new {UserId = 2, Street = "Street2", City = "city2"});
}
```

Как видно из листинга 9.2, вы должны определить первичный ключ, даже если обычно он генерируется базой данных, чтобы можно было определять связи, устанавливая во внешние ключи соответствующие значения первичных ключей. А если вы измените первичный ключ, то предыдущая заполненная запись будет удалена. Кроме того, если вы сохраните исходный первичный ключ, но измените данные в этой записи, то миграция обновит ее.

> **ПРИМЕЧАНИЕ** Каталог Chapter09Listings\SeedExample из проекта Test связанного репозитория GitHub содержит пример того, что происходит, когда вы меняете исходные данные между миграциями. Вторая миграция содержит код для удаления, обновления и вставки исходных данных вследствие изменений в частях метода `HasData`.

9.4.4 Миграции и несколько разработчиков

Когда над проектом, в котором используются миграции, работают несколько разработчиков, вы можете столкнуться со слиянием кода программы, при котором миграция одного из разработчиков конфликтует с вашей. В этом разделе приводится несколько советов, что делать в такой ситуации. Предполагаю, что вы используете систему контроля версий и что у вас есть собственная база данных для разработки, чтобы можно было опробовать миграцию локально.

Во-первых, если новая миграция не конфликтует с миграцией, которая уже добавлена в приложение, у вас не должно быть конфликта системы контроля версий, потому что миграции EF Core спроектированы по типу team-friendly (в отличие от миграций EF6).

Миграции могут быть применены в несколько ином порядке; возможно, вы создали свою миграцию вчера, а чья-то еще миграция была создана сегодня и применена к основной базе данных. Такая ситуация не должна вызывать проблем, если конфликтов слияния нет, потому что EF Core умеет справляться с неупорядоченными миграциями.

Вы узнаете, есть ли у вас конфликт слияния миграций, потому что ваша система контроля версий покажет конфликт в файле снимка миграции <DbContextClassName>ModelSnapShot. Если такой конфликт произойдет, вот рекомендуемый способ исправить ситуацию:

1. Прервите слияние системы контроля версий, содержащее изменение, противоречащее вашей миграции.

2. Удалите созданную вами миграцию, используя любую из следующих команд.
   (*Примечание*: сохраните классы сущностей и изменения конфигурации – они вам понадобятся позже:
   - CLI: `dotnet ef migrations remove`;
   - PMC: `Remove-Migration`.)
3. Примените слияние с входящей миграцией, от которого вы отказались на этапе 1. Конфликт слияния больше не должен возникать в файле снимка миграции.
4. Используйте команду `add migration`, чтобы воссоздать свою миграцию.

Этот процесс разрешения конфликтов миграций работает в большинстве случаев, но может оказаться сложным. Вот что я рекомендую для проектов, в которых могут возникать конфликты миграций:

- перед созданием миграции осуществите слияние основной/промышленной ветки в локальную;
- сделайте только одну миграцию в системе контроля версий, объединенную с основной/промышленной веткой, потому что отмена двух миграций – тяжелая работа;
- сообщите членам своей команды разработчиков, если вы считаете, что миграция может повлиять на их работу.

9.4.5 Использование собственной таблицы миграций, позволяющей использовать несколько DbContext в одной базе данных

EF Core создает служебную таблицу, если вы применяете миграцию к базе данных. Он использует эту таблицу, чтобы отделять уже примененные миграции от тех, которые следует применить к базе данных. По умолчанию эта таблица называется __EFMigrationsHistory, но с помощью метода `MigrationsHistoryTable` имя можно изменить.

Причин для изменения таблицы истории миграций не так много, но одна из них – совместное использование базы данных в нескольких DbContext. Вот два примера:

- экономия средств за счет объединения баз данных – вы создаете приложение ASP.NET Core с отдельными учетными записями пользователей, которому требуется база данных учетных записей. DbContext приложения также нужна база данных. Использование собственной таблицы миграций позволит обоим контекстам применять одну и ту же базу данных;
- использование отдельного DbContext для каждой бизнес-группы. В третьей части книги я хочу облегчить расширение проекта по мере его роста. Поэтому использую отдельные DbContext: один для кода отображения книги и другой для кода обработки заказа.

Оба примера работают, но использование системы миграций EF Core с любым из них отнимает немного больше усилий. Первый пример – экономия средств за счет объединения баз данных – проще, потому что у двух объединяемых баз данных нет общих таблиц, представлений и прочего. Но поскольку обе базы данных используют систему миграций, им нужны разные таблицы истории миграций. База данных учетных записей пользователей ASP.NET Core использует для таблицы истории миграций имя по умолчанию, поэтому имя DbContext приложения нужно изменить. В следующем листинге показано, как это сделать, когда вы регистрируете DbContext приложения в `Startup` классе ASP.NET Core.

Листинг 9.3 Изменение имени таблицы истории миграций для DbContext

Регистрирует DbContext приложения как сервис в ASP.NET Core

Второй параметр позволяет выполнять настройку на уровне поставщика базы данных

```
services.AddDbContext<EfCoreContext>(
    options => options.UseSqlServer(connection,
        dbOptions =>
         dbOptions.MigrationsHistoryTable("NewHistoryName")));
```

Метод MigrationsHistoryTable позволяет изменить имя таблицы миграции и, при необходимости, схему таблицы

Затем, конечно, вы должны выполнить миграцию для каждого DbContext – в данном случае контекст учетных записей пользователя ASP.NET Core и DbContext приложения. На этом ваша работа окончена.

Что касается второго примера – наличия отдельного DbContext для каждой бизнес-группы, – для каждого DbContext необходимо разное имя таблицы истории миграций, чтобы каждая миграция была отдельной. Кроме того, нужно указать отдельные каталоги для классов миграций для каждого DbContext. Это можно сделать с помощью параметра в команде `add migration`. Эта команда не допустит конфликтов в именах классов, если в обоих DbContext используется одно и то же имя миграции.

> **ПРИМЕЧАНИЕ** Также при желании можно поместить классы миграции в отдельный проект. Нужно сообщить команде `add migration`, в какой проект разместить миграции. Затем используется метод `MigrationsAssembly` при настройке параметров базы данных: http://mng.bz/aonB.

Однако в этом примере есть еще одна проблема, с которой приходится иметь дело: каждому DbContext необходимо получить доступ к таблице Books, что приводит к дублированию миграции этой таблицы. Таблица Books является общей, потому что оба DbContext должны иметь возможность читать ее (чтобы показывать книги и создавать заказ на приобретение книги).

Есть несколько вариантов решения этой проблемы, но лучше всего использовать метод Fluent API `ExcludeFromMigrations`, который запрещает включение данного класса сущности в миграцию. В примере с `BookDbContext/OrderDbContext` можно удалить миграцию класса сущности `Book` в `OrderDbContext`, как показано в этом фрагменте кода:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>()
        .ToTable("Books",
            t => t.ExcludeFromMigrations());
}
```

Если класс сущности `Book` отображается в представление, а не в таблицу (см. раздел 7.9.3), то инструменты миграции не будут включать это представление в миграцию. Для данного примера это хороший подход, поскольку мы хотим, чтобы у `BookDbContext` был доступ на чтение и запись, а у `OrderDbContext` – только на чтение.

Мой подход к созданию миграций

Мой подход к созданию миграций основан на наличии модульных тестов, которые могут что-то проверить в базе данных. Я понимаю, что некоторым разработчикам не нравится такой подход, но я обнаружил, что когда у меня нет возможности провести модульное тестирование своего кода с реальной базой данных, мне нужно создать и применить миграцию, а затем запустить приложение для проверки изменений. Использование модульных тестов с реальной базой данных ускоряет разработку, и каждый модульный тест, который я пишу, улучшает охват тестов приложения, над которым я работаю.

Обычно я создаю исчерпывающий набор модульных тестов для всего приложения, кроме UI/WebAPI. Многие из моих модульных тестов используют базу данных, потому что это самый быстрый способ настроить тестовые данные; EF Core с легкостью выполняет настройку тестовой базы данных. Конечно, для сложной бизнес-логики я использую паттерн «Репозиторий» (см. раздел 4.2), для которого можно сделать заглушку (stub), но для простых запросов и обновлений я могу использовать тестовые базы данных. В результате можно поэтапно внедрять новую функциональную возможность и проверять ее по мере необходимости, запуская модульные тесты.

Этот подход требует, чтобы базы данных в модульных тестах находились в актуальном состоянии с текущей моделью EF Core; схема должна соответствовать текущим классам сущностей и конфигурации DbContext. Многолетний опыт (и некоторые предложения от команды EF Core) отточили мой подход, которым я поделюсь с вами в главе 17. Такой подход позволяет создавать сложную функциональность с меньшим количеством этапов, при этом базы данных модульного тестирования всегда «идут в ногу» с текущей моделью EF Core. Только после того, как весь код написан и модульные тесты пройдены, я наконец создаю миграцию.

9.5 Редактирование миграции для обработки сложных ситуаций

Инструменты миграции EF Core мощны и хорошо продуманы, но они не могут справиться с любой возможной миграцией базы данных, например критическим изменением, приводящим к потере данных. Команде EF Core это известно, поэтому она предоставила несколько способов изменять класс миграции вручную. Посмотрим на типы миграции, с которыми стандартная миграция не справится без посторонней помощи:

- *критические изменения, приводящие к потере данных*, например перемещение столбцов из одной таблицы в новую;
- *добавление функциональных возможностей SQL, которые не создает EF Core*, например добавление пользовательских функций, хранимые процедуры SQL, представления и т. д.;
- *изменение миграции для работы с несколькими типами баз данных*, например работа с SQL Server и PostgreSQL.

Можно исправить эти проблемы, отредактировав стандартный класс миграции, созданный с помощью команды `add migration`. Для этого необходимо отредактировать класс миграции, имя файла которого заканчивается именем миграции и который имеет расширение .cs, например ..._InitialMigration.cs. В следующих разделах вы узнаете о различных вариантах редактирования, которые могут улучшить или исправить ваши миграции, а в табл. 9.2 приводится обзор плюсов и минусов ручного редактирования миграции для достижения желаемого результата.

Таблица 9.2 Обзор положительных и отрицательных сторон и ограничений миграции, созданной командой `add migration` и отредактированной вами, чтобы справиться с ситуациями, с которыми стандартная миграция не справляется своими силами

	Примечания
Положительные стороны	▪ Вы начинаете по большей части с миграций, созданных с помощью команды `add migration`. ▪ Вы можете настроить миграцию под себя. ▪ Вы можете добавить дополнительные функции SQL, например хранимые процедуры
Отрицательные стороны	▪ Вам нужно больше знать о структуре базы данных. ▪ Некоторые правки требуют знания SQL
Ограничения	Ваши правки не проверяются EF Core, поэтому вы можете получить несоответствие между обновленной базой данных и классами сущностей и DbContext приложения
Советы	То же, что и для стандартных миграций (см. табл. 9.1)
Мой вердикт	Этот подход отлично подходит для небольших изменений, но внесение серьезных изменений может стать тяжелой работой, поскольку вы часто смешиваете C#-код с SQL. Если вы планируете редактировать большое количество миграций, чтобы добавить функциональные возможности SQL, то в качестве альтернативы следует рассмотреть подход с использованием сценария SQL (см. раздел 9.6.2)

9.5.1 Добавление и удаление методов MigrationBuilder внутри класса миграции

Начнем с простого примера исправления миграции, которая содержит критическое изменение, приводящее к потере данных. В этом примере показано, что произойдет, если изменить имя свойства в классе сущности. Эту проблему можно исправить, удалив две команды и заменив их на метод `RenameColumn` из класса `MigrationBuilder` внутри класса миграции.

Данный пример взят из главы 7, где мы изменили свойство `CustomerId` из класса сущности `Order` на `UserId`, чтобы автоматизировать добавление фильтра запросов (см. раздел 7.15.4). Стандартная миграция рассматривает эту операцию как удаление свойства `CustomerId` и добавление нового свойства `UserId`, что приведет к потере всех существующих значений в столбце `CustomerId`. Чтобы решить эту проблему, выполните следующие изменения в классе миграции, сгенерированные стандартной миграцией из главы 7:

- удалите команду `AddColumn`, которая добавляет новый столбец `UserId`;
- удалите команду `DropColumn`, которая удаляет существующий столбец `CustomerId`;
- добавьте команду `RenameColumn`, чтобы переименовать столбец `CustomerId` в `UserId`.

В следующем листинге показан измененный класс миграции, имя которого взято из имени миграции, `Chapter07`. Методы, которые нужно удалить, закомментированы, и добавлен новый метод `RenameColumn`.

Листинг 9.4 Обновленный класс миграции. Старые команды заменены

Класс миграции, созданный командой add migration, который был отредактирован

```
public partial class Chapter07 : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        //migrationBuilder.AddColumn<Guid>(
        // name: "UserId",
        // table: "Orders",
        // type: "uniqueidentifier",
        // nullable: false,
        // defaultValue:
        //     new Guid("00000000-0000-0000-0000-000000000000"));

        //migrationBuilder.DropColumn(
        // name: "CustomerId",
        // table: "Orders");
```

В классе миграции есть два метода. Метод Up применит миграцию, а метод Down удалит ее

Команда добавления нового столбца UserId не должна выполняться, поэтому закомментируем ее

Команда удаления существующего столбца CustomerId не должна выполняться, поэтому закомментируем ее

```
        migrationBuilder.RenameColumn(
            name: "CustomerId",
            table: "Orders",
            newName: "UserId");

        //… Остальная часть кода опущена;
    }
}
```

Правильный подход – переименовать столбец CustomerId в UserId

Этот код изменяет метод миграции `Up`, в результате чего вместо потери данных данные, содержащиеся в старом столбце `CustomerId`, будут сохранены. Класс миграции, созданный командой `add migration`, также содержит метод `Down`, который отменяет миграцию, если к базе данных была применена миграция `Up` (см. команду `remove` в разделе 9.4.4). Поэтому рекомендуется отредактировать метод `Down` для правильной отмены миграции. Часть, относящаяся к методу `Down`, из листинга 9.4 также будет отредактирована, чтобы выполнить действие, обратное части, относящейся к методу `Up`. Мы бы удалили команды `AddColumn` и `DropColumn` в части `Down` и заменили их на `RenameColumn`, но теперь `UserId` переименован обратно в `CustomerId`.

ПРИМЕЧАНИЕ Я не привожу здесь измененный метод `Down`, но вы можете найти этот класс миграции в папке Migrations проекта DataLayer в репозитории GitHub, ветка `Part2`.

9.5.2 Добавление команд SQL в миграцию

Для добавления команд SQL в миграцию может быть две основные причины: критическое изменение, приводящее к потере данных, а также добавление или изменение частей базы данных SQL, которые EF Core не контролирует, например добавление представлений или хранимых процедур.

В качестве примера добавления команд SQL к миграции мы займемся критическим изменением, приводящим к потере данных. В данном случае мы начнем с базы данных с классом сущности `User`, который содержит имя каждого пользователя и его адрес в свойствах `Street` и `City`. По мере продвижения проекта мы решили, что хотим скопировать адресную часть в другую таблицу, и пусть класс сущности `User` ссылается на нее через навигационное свойство. На рис. 9.4 показаны состояния схемы базы данных до и после и содержимое таблиц.

Лучший способ справиться с этой ситуацией, используя миграции EF Core, – добавить команды SQL для копирования данных, но это нетривиальный процесс. Изменение миграции требует добавления кода SQL.

ПРИМЕЧАНИЕ Всю миграцию можно увидеть в соответствующем репозитории GitHub на странице http://mng.bz/goME.

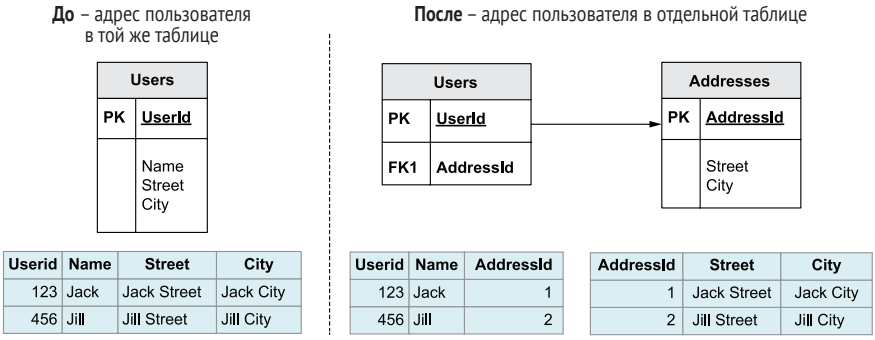


Рис. 9.4 Исходная схема (до) базы данных и данные с одной таблицей Users. В новой схеме базы данных (после) есть новая таблица Addresses, а адресные данные из исходной таблицы Users были перенесены в таблицу Addresses. Кроме того, были удалены адресные столбцы таблицы Users, Street и City и был добавлен новый внешний ключ `AddressId` для связи с адресом пользователя

Вначале мы изменяем класс сущности `User`, чтобы удалить адрес и добавить ссылку на новый класс сущности `Address` в DbContext. Затем мы создаем новую миграцию, используя команду `add migration`, которая предупредит нас, что это может привести к потере данных. Теперь мы готовы редактировать миграцию.

Второй этап – добавление серии команд SQL с помощью метода `MigrationBuilder.Sql`, например `migrationBuilder.Sql("ALTER TABLE…")`. В следующем листинге показаны команды SQL без `migrationBuilder.Sql`, чтобы их было легче рассмотреть.

Листинг 9.5 Команды SQL Server для копирования адресов в новую таблицу

```
ALTER TABLE [Addresses]
    ADD [UserId] [int] NOT NULL

INSERT INTO [Addresses] ([UserId],[Street],[City])
    SELECT [UserId],[Street],[City] FROM [Users]

UPDATE [Users] SET [AddressId] = (
    SELECT [AddressId]
        FROM [Addresses]
        WHERE [Addresses].[UserId] = [Users].[Userid])

ALTER TABLE [Addresses]
   DROP COLUMN [UserId]
```

Добавляет временный столбец, позволяющий установить правильный внешний ключ в таблице Users

Копирует все адресные данные с первичным ключом пользователя в таблицу Addresses

Использует временный столбец UserId, чтобы убедиться, что установлены правильные внешние ключи

Устанавливает внешний ключ в таблице Users для связи с таблицей Addresses

Удаляет временный столбец UserId из таблицы Addresses, так как он больше не нужен

Мы добавляем эти SQL-команды в миграцию с помощью метода `migrationBuilder.Sql` для каждой команды, помещая их после создания таблицы Addresses, но до настройки внешнего ключа. Кроме того, методы `MigrationBuilder`, которые удаляют адресные свойства из таб-

лицы Users, должны быть расположены после выполнения кода SQL; в противном случае данные будут утеряны до того, как SQL сможет их скопировать.

> **ПРИМЕЧАНИЕ** В разделе 9.8.1 описывается способ выполнения кода C# до и после применения конкретной миграции к базе данных. Такой подход – еще один способ копировать данные, но часто подход с SQL работает лучше.

9.5.3 Добавление собственных команд миграции

Если вы часто добавляете определенные типы SQL-команд к миграции, то можно создать некий шаблонный код, чтобы упростить редактирование. Создание шаблонов, например добавление представления SQL в базу данных, – неплохая идея, если вы нередко используете функциональность SQL, потому что затраты на создание шаблона меньше, чем многократное написание кода этой функциональности вручную. Создать шаблон можно двумя способами:

- создать методы расширения, принимающие класс `MigrationBuilder` и создающие команды с помощью метода `Sql` класса `MigrationBuilder`. Эти методы расширения, как правило, зависят от базы данных;
- более сложный, но более универсальный подход – расширить класс `MigrationBuilder` для добавления собственных команд. Этот подход позволяет получить доступ к методам для создания команд, которые подходят для многих поставщиков баз данных.

В данном разделе обсуждается только первый подход. Второй подход – продвинутая версия, прекрасно описанная в документации по EF Core на странице http://mng.bz/xGBe.

В качестве примера мы создадим метод расширения, что облегчит создание представления SQL. Метод расширения принимает класс, который будет отображаться в представление, чтобы он мог найти свойства для отображения в столбцы (при условии что мы используем только свойства и именование столбцов по соглашению). В следующем листинге показан метод расширения, создающий представление в рамках миграции.

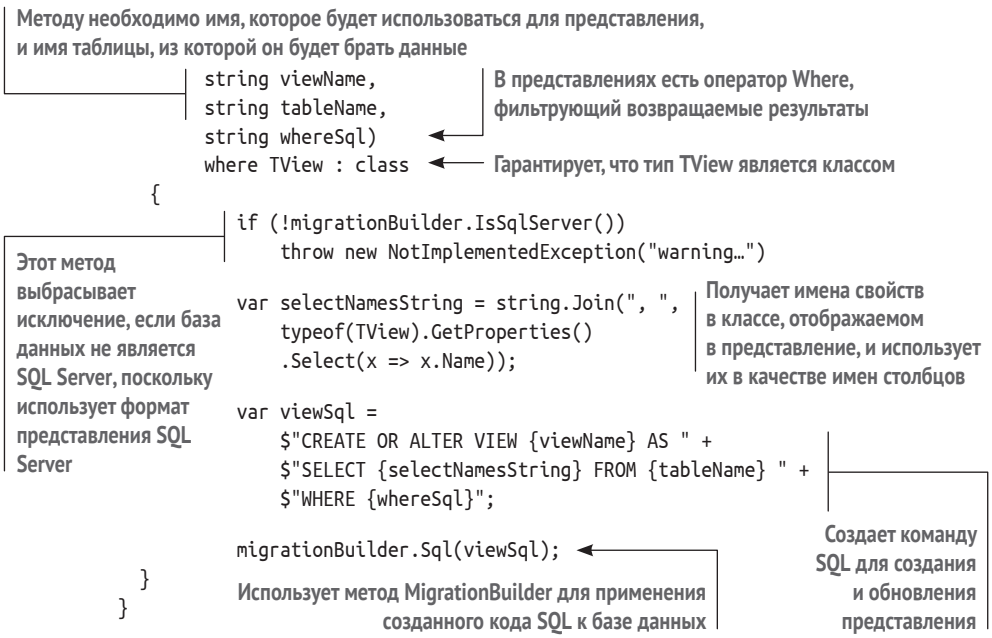
Листинг 9.6 Метод расширения для добавления и изменения представления SQL в миграции EF Core

Метод расширения должен быть в статическом классе

Этому методу нужен класс, который отображается в представление, чтобы можно было получить свойства

```
public static class AddViewExtensions
{
    public static void AddViewViaSql<TView>(
        this MigrationBuilder migrationBuilder,
```

Класс MigrationBuilder предоставляет доступ к методам миграции – в данном случае к методу Sql



Этот метод можно использовать при миграции, добавив его к методу `Up` (и команду `DROP VIEW` в методе `Down`, чтобы удалить представление). Вот фрагмент кода, создающий представление для класса `MyView`, у которого есть свойства `MyString` и `MyDateTime`:

```
migrationBuilder.AddViewViaSql<MyView>(
    "EntityFilterView", "Entities",
    "MyDateTime >= '2020-1-1'");
```

Получившийся в итоге код выглядит следующим образом:

```
CREATE OR ALTER VIEW EntityFilterView AS
SELECT MyString, MyDateTime
FROM Entities
WHERE MyDateTime >= '2020-1-1'
```

9.5.4 Изменение миграции для работы с несколькими типами баз данных

Миграции EF Core зависят от поставщика базы данных, т. е. если вы создаете миграцию для SQL Server, она почти наверняка не подойдет для базы данных PostgreSQL. Однако миграция для нескольких типов баз данных требуется редко. Вообще, я не рекомендую использовать несколько типов баз данных с одним и тем же кодом EF Core, поскольку между разными типами баз данных существуют тонкие различия, на чем можно попасться (см. главу 16). Но если вам нужно поддерживать миграции для двух или более типов баз данных, то рекомендуется создавать отдельные миграции для каждого поставщика базы

данных. Например, если вы хотите использовать базу данных SQLite для Linux – версии вашего приложения и базу данных SQL Server для Windows, то нужно будет выполнить следующие шаги.

Первый этап – создание отдельного DbContext для каждого типа базы данных. Самый легкий способ – создать основной DbContext приложения и наследоваться от него в DbContext конкретной базы данных. В следующем листинге показаны DbContext двух приложений, причем второй наследуется от первого.

Листинг 9.7 Два DbContext с одинаковыми классами сущностей и конфигурацией

```
public class MySqlServerDbContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    // … Остальные DbSet опущены;

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
     //… Здесь идет код Fluent API;
    }
}
public class MySqliteDbContext : MySqlServerDbContext
{
}
```

Наследует обычный класс DbContext

Добавляет все свойства DbSet и Fluent API, которые используются в базах данных обоих типов

MySqliteDbContext наследует класс MySqlServerDbContext вместо обычного DbContext

MySqliteDbContext наследует свойства DbSet и Fluent API от MySqlServerDbContext

Следующий шаг – создание способа, с помощью которого инструменты миграции могут получить доступ к каждому DbContext с определенным поставщиком базы данных. Самый чистый способ – создать класс `IDesignTimeDbContextFactory<TContext>`, как описано в разделе 9.4.1. В качестве альтернативы можно переопределить метод `OnConfiguring` в каждом DbContext для определения поставщика базы данных.

На данном этапе можно создать миграцию для каждого типа базы данных с помощью команды `AddMigration` (см. раздел 9.4.2). Важный момент: каждая миграция должна находиться в отдельном проекте, чтобы при создании миграции можно было получить доступ к нужным классам миграции для получения типа базы данных, с которой связан DbContext. Мы сообщаем EF Core, где можно найти классы миграции, используя метод `MigrationsAssembly` при создании варианта базы данных. В следующем фрагменте кода показан метод `AddDbContext`, используемый для регистрации DbContext приложения, в котором устанавливаются поставщик базы данных и проект `Database.SqlServer` в качестве источника миграций для базы данных:

```
services.AddDbContext<MySqlServerDbContext>(
    options => options.UseSqlServer(connection,
       x => x.MigrationsAssembly("Database.SqlServer")));
```

В качестве альтернативы можно использовать одну миграцию и добавить код `if/then` внутри нее, чтобы изменить действия миграции в зависимости от поставщика базы данных. Данный подход не рекомендуется, потому что его сложнее поддерживать. Если вам нужна дополнительная информация об этом подходе, предлагаю взглянуть на документацию EF Core, где описаны оба подхода (http://mng.bz/pV08).

> **ПРИМЕЧАНИЕ** Cosmos DB и нереляционные базы данных в целом не используют миграции EF Core, потому что у них нет фиксированной схемы, как у реляционных баз данных, и обычно их миграция происходит с помощью сценария обновления. У миграции базы данных Cosmos DB, доступ к которой осуществляется через EF Core, имеется ряд проблем, которые обсуждаются в главе 16.

9.6 Использование сценариев SQL для создания миграций

Следующий способ управления изменением схемы базы данных – создание *сценариев изменения* SQL и последующее их применение к любой из ваших баз данных. Сценарии изменений содержат команды SQL, которые обновляют схему базы данных. Такой подход к работе с обновлениями схемы базы данных является более традиционным и предоставляет гораздо лучший контроль над функциональностью базы данных и обновлением схемы. Необходимо хорошее знание команд SQL, чтобы писать и понимать эти сценарии миграции, но инструменты могут создавать их за вас, сравнивая базы данных.

Как и в случае с миграциями, которые может создать EF Core, ваша цель – создать миграцию, которая изменит схему базы данных в соответствии с ее внутренней моделью. В этом разделе мы рассмотрим два подхода:

- использование инструментов сравнения баз данных SQL для выполнения миграции из текущей схемы базы данных в желаемую;
- написание кода сценария изменения для миграции базы данных вручную.

Первый вариант должен производить сценарий для полного соответствия внутренней модели базы данных, тогда как во втором варианте разработчик должен написать правильный код SQL, соответствующий требованиям EF Core. Если разработчик ошибется (и могу засвидетельствовать, что это не трудно), то приложение может дать сбой, выбросив исключение; хуже того, данные могут незаметно потеряться. В конце этого раздела я приведу описание созданного мной инструмента, который сравнивает схему базы данных с текущей моделью базы данных EF Core и сообщает о наличии различий.

9.6.1 Использование инструментов сравнения баз данных SQL для выполнения миграции

Один из подходов к созданию сценария изменения на SQL – сравнить две базы данных: свою исходную базу данных и новую, созданную EF Core после обновления конфигурации. Инструменты могут сравнить две базы данных и показать различия в схемах. Многие из этих инструментов сравнения также могут создать сценарий, который изменит схему исходной базы данных на схему базы данных, в которую вы хотите мигрировать. Поэтому если вы можете создать базу данных с нужной схемой, то инструмент сравнения поможет создать SQL-сценарий изменения, необходимый для того, чтобы обновить базу данных до требуемой. Инструменты сравнения позволяют с легкостью создавать сценарии изменения на SQL, но, как и у всего остального, у них есть свои особенности. Прежде чем рассматривать детали, посмотрите на табл. 9.3, где приводится обзор этого подхода.

Для многих типов серверов баз данных доступно несколько инструментов сравнения с открытым исходным кодом и коммерческих инструментов; они могут сравнивать схемы базы данных и выводить сценарии изменения. В этом примере используется обозреватель объектов SQL Server, встроенный в Visual Studio (любой версии), который можно найти в рабочей нагрузке Data Storage and Processing (Хранение и обработка данных) установщика Visual Studio. Можно получить инструмент напрямую, выбрав **Tools** > **SQL Server** > **New Schema Comparison** (Инструменты > SQL Server > Новое сравнение схем).

Таблица 9.3 Краткий обзор достоинств, недостатков и ограничений использования инструмента сравнения баз данных SQL для создания сценариев изменения для миграции базы данных

	Примечания
Положительные стороны	▪ Инструменты создают корректный сценарий миграции SQL за вас
Отрицательные стороны	▪ Необходимо понимание баз данных и SQL. ▪ Инструменты сравнения часто выводят все возможные параметры, чтобы убедиться, что они все делают правильно, а это затрудняет понимание вывода кода SQL. ▪ Не все инструменты сравнения SQL создают сценарий удаления миграции
Ограничения	▪ Инструменты не обрабатывают критические изменения, поэтому требуют участия человека
Советы	Я использую этот подход только для сложных и больших миграций и убираю все лишние настройки, чтобы упростить работу с кодом
Мой вердикт	Это полезный подход, он особенно хорош для тех, кто плохо знаком с языком SQL. Кроме того, он полезен для тех, кто пишет собственный SQL-код миграции и хочет убедиться, что этот код корректен

ПРИМЕЧАНИЕ Пошаговое руководство по использованию обозревателя объектов SQL Server можно найти на странице http://mng.bz/OEDR.

На рис. 9.5 показано, как сравнить базу данных из главы 2 с изменениями в главе 4, где мы добавляем классы сущностей `Order` и `LineItem`. Инструмент сравнения SQL полагается на наличие двух баз данных:

- первая база данных – это текущее состояние базы данных, известное как исходная база данных. Нужно выполнить обновление до новой схемы, которая отображается как Chapter02Db на рис. 9.5. Чаще всего это ваша рабочая база данных или какая-то другая база, соответствующая исходной схеме;
- вторая база данных, известная как целевая, должна иметь схему, до которой вы хотите обновиться. На рис. 9.5 она показана как Chapter04Db.Test. Эта база данных, скорее всего, находится в окружении разработки. Есть одна прекрасная функция, которую я использую для получения такой базы данных, – это метод EF Core `EnsureCreated`. Этот метод, обычно используемый в модульном тестировании, создает базу данных на основе текущих классов сущности и конфигурации EF Core.

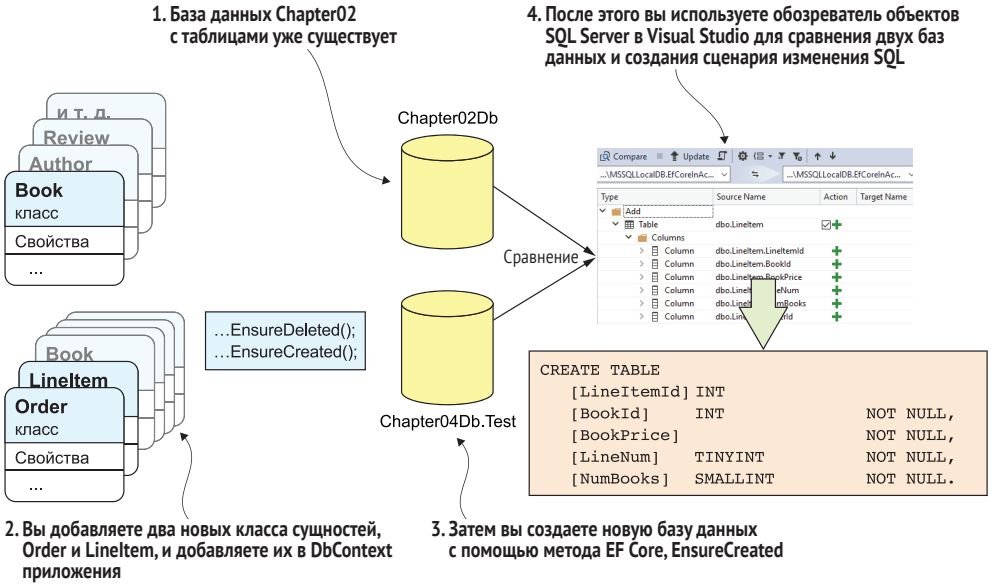


Рис. 9.5 Процесс создания сценария изменения SQL путем сравнения двух баз данных. Важный момент – вторая база данных, Chapter04Db.Test, создается EF Core, поэтому вы знаете, что она соответствует текущей модели EF Core. В этом примере используется Обозреватель объектов SQL Server из Visual Studio для сравнения двух баз данных и создания сценария изменений, который перенесет базу данных Chapter02 на правильную схему, содержащую изменения из главы 4

Настроив обе базы данных в качестве исходной и целевой баз данных в инструменте сравнения схем SQL, можно сравнить две схемы и затем создать сценарий изменения, который преобразует схему исходной базы данных в схему целевой базы данных.

Этот процесс изначально говорит нам о различиях; тогда у нас есть возможность создать сценарий изменения SQL, который перенесет базу данных из схемы исходной базы данных в нужную схему. При таком варианте создается SQL-сценарий изменения, который перенесет базу данных из схемы исходной базы, Chapter02Db на рис. 9.5, в схему целевой базы Chapter04Db.Test на рис. 9.5. Я расскажу, как применять сценарий изменения, в разделе 9.8.4.

9.6.2 Написание кода сценариев изменения SQL для миграции базы данных вручную

Еще один подход – самостоятельно создавать команды SQL, необходимые для миграции. Этот вариант привлекателен для разработчиков, которые хотят определить содержимое базы данных способами, недоступными в EF Core. Можно использовать этот подход, чтобы установить более строгие ограничения CHECK для столбцов, добавлять хранимые процедуры или определяемые пользователем функции и т. д. с помощью сценариев SQL.

Единственный минус для разработчика состоит в том, что нужно достаточно хорошо знать SQL для написания и редактирования сценариев изменения. Это требование может оттолкнуть некоторых разработчиков, но все не так плохо, как вы думаете, потому что можно просмотреть результирующий SQL в EF Core, чтобы создать базу данных, а затем скорректировать этот код, используя свои изменения. В табл. 9.4 приведен обзор данного подхода.

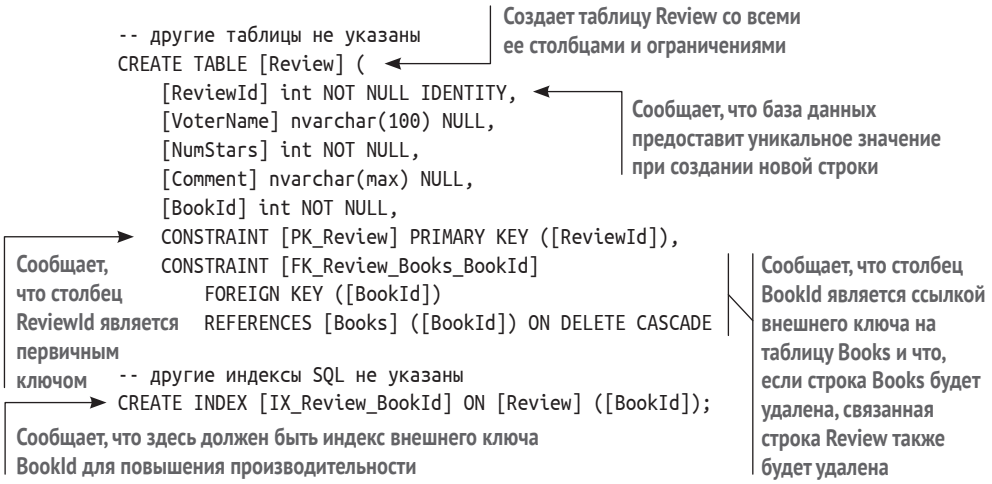
Таблица 9.4 Краткий обзор плюсов, минусов и ограничений написания кода сценариев изменения SQL вручную для миграции базы данных

	Примечания
Положительные стороны	▪ У вас есть полный контроль над структурой базы данных, в том числе и части, которая не добавляет EF Core, например определяемые пользователем функции и ограничения столбцов
Отрицательные стороны	▪ Необходимо понимать команды SQL, например CREATE TABLE. ▪ Вы должны сами определить, какие изменения произошли (но посмотрите строку «Советы»). ▪ Сценария для автоматического удаления миграции не существует. ▪ Данный подход не гарантирует создания корректной миграции (но см. CompareEfSql в разделе 9.6.3)
Ограничения	Никаких
Советы	Можно использовать команду миграции `Script-DbContext`, чтобы получить фактический код SQL, который будет выводить EF Core, а затем искать отличия в SQL от предыдущей схемы базы данных, что значительно упрощает написание миграции
Мой вердикт	Этот подход предназначен для тех, кто знает SQL и хочет получить полный контроль над базой данных. Безусловно, это заставляет думать о наиболее оптимальных настройках для своей базы данных, которые могут улучшить производительность

Создание сценария SQL упрощается за счет команды `scriptdbcontext`, выводящей команды SQL, которые EF Core будет использовать

для создания новой базы данных (эквивалент вызова метода `context.Database.EnsureCreated`). В следующем листинге показана небольшая часть кода SQL, созданного методом `EnsureCreated`, с акцентом на таблицу Review и ее индексы.

Листинг 9.8 Часть кода SQL, сгенерированного методом EnsureCreated при создании базы данных



Поскольку вы знаете, какие классы сущностей конфигураций EF Core вы изменили, то можете найти соответствующую часть кода SQL, которая должна отражать эти изменения. Эта информация должна помочь вам написать команду SQL, и вы с большей вероятностью напишете сценарии изменений, соответствующие ожиданиям EF Core.

Как и в случае миграции EF Core, вы создаете серию сценариев изменения, которые нужно применить к вашей базе данных в определенном порядке. Чтобы облегчить этот процесс, следует дать своим сценариям имена, определяющие порядок, например это может быть число или сортируемая дата.

Вот примеры имен сценариев SQL, которые я использовал для клиентского проекта:

```
Script001 - Create DatabaseRegions.sql
Script002 - Create Tenant table.sql
Script003 - TenantAddress table.sql
Script004 - AccountingCalenders table.sql
```

Сценарии не только должны применяться к базе данных в определенном порядке, но и сделать это нужно только один раз; я расскажу об этом в разделе 9.8.

Следует ли писать код удаления миграции для сценариев изменения SQL?

Миграции EF Core создают методы миграции `Up` и `Down`. Метод `Down`, известный как *обратная миграция*, содержит код для отката миграции `Up`. Некоторые разработчики, переходящие на сценарии изменения SQL, беспокоятся об отсутствии функции удаления миграции.

И хотя возможность удалить миграцию – это замечательно, скорее всего, вы не будете часто ей пользоваться. EF Core может автоматически создать миграцию, которая откатывает другую миграцию, но когда дело доходит до сценариев изменения SQL, создание сценария с `Down` не происходит автоматически, поэтому если он вам нужен, вы должны написать его самостоятельно.

Поэтому я создаю миграцию с удалением только в том случае, если мне это нужно, так как любая такая миграция – это еще один новый сценарий изменения, который откатывает последнюю миграцию. Но имейте в виду: я делаю это только после интенсивного тестирования своих миграций задолго до промышленной эксплуатации, потому что необходимость писать сценарий миграции `Down` из-за того, что ваша рабочая система не работает из-за неправильной миграции, несколько утомительна!

9.6.3 Проверка соответствия сценариев изменения SQL модели базы данных EF Core

Я использовал написанные вручную сценарии изменения в нескольких проектах, как в EF6, так и в EF Core, и основная проблема заключается в том, чтобы убедиться, что мои изменения в базе данных соответствуют модели базы данных EF Core. Поэтому я создал инструмент под названием EfSchemaCompare, который сравнивает модель базы данных EF Core со схемой реальной базы данных. Хотя у него есть некоторые ограничения, он дает хорошую обратную связь о различиях между перенесенной базой данных и моделью базы данных EF Core.

ПРИМЕЧАНИЕ Я рассматриваю модель базы данных EF Core, доступ к которой можно получить через свойство `Model` в DbContext приложения в главе 11.

На рис. 9.6 показано, как EfSchemaCompare сравнивает базу данных, которая была обновлена сценариями изменения SQL на соответствие модели базы данных EF Core.

EfSchemaCompare доступен в моей библиотеке EfCore.SchemaCompare (http://mng.bz/Yq2B). С помощью этого инструмента я создаю модульные тесты, которые проверяют мою базу данных, используемую для разработки, – и, что более важно, мою рабочую базу данных, –

чтобы увидеть, нет ли расхождений в модели базы данных EF Core и реальной схемы.

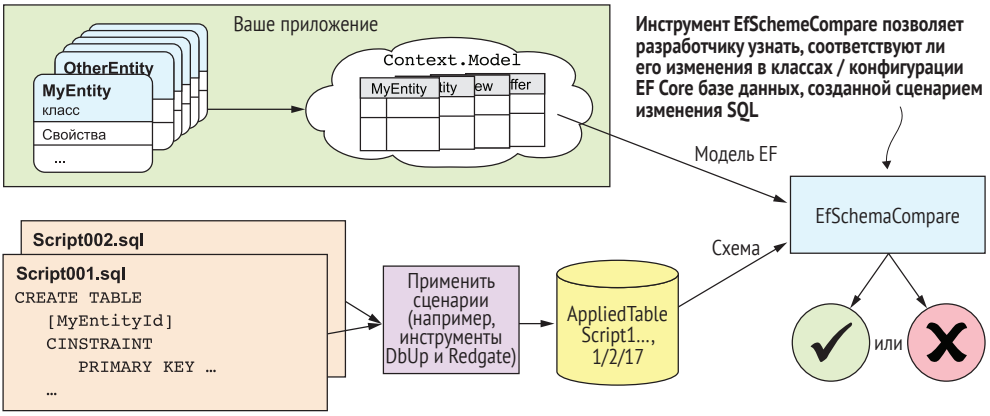


Рис. 9.6 EfSchemaCompare сравнивает модель базы данных EF Core, которую он формирует, просматривая классы сущностей и конфигурацию DbContext приложения, со схемой базы данных, которая была обновлена с помощью сценариев изменения SQL. Если обнаруживается разница – выводятся удобные для восприятия сообщения об ошибках

9.7 Использование инструмента обратного проектирования EF Core

В некоторых случаях у вас уже есть база данных, к которой вы хотите получить доступ через EF Core. Для этого необходимо применить обратные миграции и разрешить EF Core создать классы сущностей и DbContext приложения, используя существующую базу данных в качестве шаблона. Этот процесс известен как обратное проектирование базы данных. Такой подход подразумевает, что база данных является источником истины. Инструмент обратного проектирования EF Core, также известный как скаффолдинг, используется для воссоздания классов сущностей и DbContext приложения со всеми необходимыми конфигурациями. В табл. 9.5 приводится обзор этого подхода, а сам процесс показан на рис. 9.7.

Этот подход в основном используется, когда вы хотите создать приложение EF Core на основе существующей базы данных, но я также опишу способ управления миграциями. Для начала посмотрим, как запустить инструмент обратного проектирования. Есть два варианта:

- запустить его через командную строку;
- использовать расширение EF Core Power Tools для Visual Studio.

Таблица 9.5 Обзор плюсов, минусов и ограничений обратного проектирования базы данных как способа доступа к существующей базе данных или постоянного обновления классов сущности и DbContext приложения, чтобы соответствовать измененной базе данных

	Примечание
Положительные стороны	▪ Инструмент создает код/классы EF Core из существующей базы данных. ▪ Инструмент позволяет сделать базу данных источником истины, а код EF Core и классы создаются и обновляются по мере изменения схемы базы данных
Отрицательные стороны	▪ Классы сущностей нельзя с легкостью отредактировать, например чтобы изменить способ реализации навигационных свойств коллекций. Но в разделе 9.7.2 приводится решение этой проблемы. ▪ Инструмент всегда добавляет навигационные ссылки на обоих концах связи (см. раздел 8.2)
Ограничения	Никаких
Советы	Если вы собираетесь неоднократно прибегать к обратному проектированию базы данных, то рекомендую использовать расширение Visual Studio EF Core Power Tools, поскольку оно запоминает настройку из последнего использования обратного проектирования
Мой вердикт	Если у вас есть существующая база данных, к которой вам нужно получить доступ через EF Core, то обратное проектирование сэкономит вам много времени

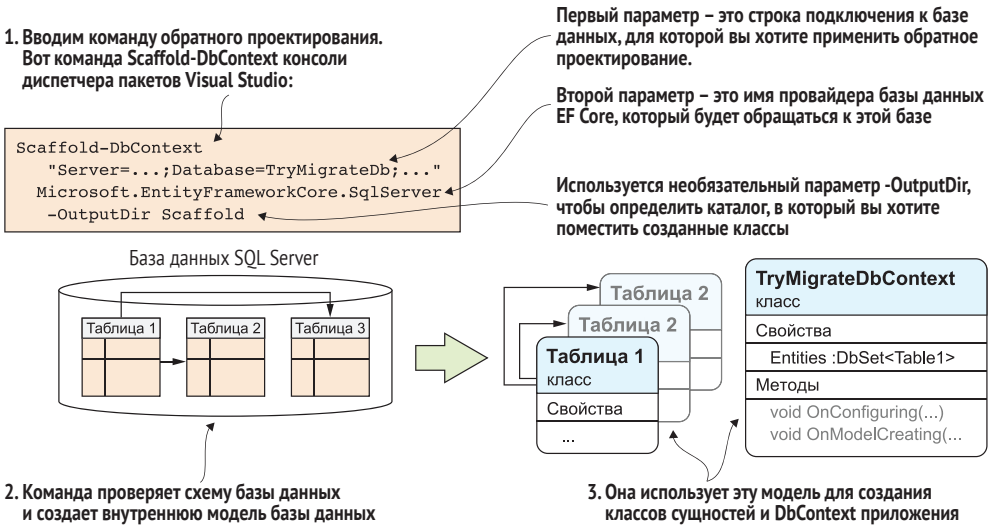


Рис. 9.7 Типичное использование команды обратного проектирования EF Core, которая проверяет базу данных, найденную через строку подключения к базе данных, а затем генерирует классы сущностей и DbContext приложения для сопоставления с базой данных. Команда использует связи с внешним ключом для построения полностью определенной связи между классами сущностей

9.7.1 Запуск команды обратного проектирования

Можно применить обратное проектирование к базе данных из командной строки (инструменты интерфейса командной строки –

CLI) или окно PMC Visual Studio. У интерфейса командной строки и PMC разные имена и параметры. В следующем списке показана команда `scaffold` для обратного проектирования базы данных приложения Book App. Обратите внимание, что эти команды выполняются в каталоге проекта ASP.NET Core Visual Studio и что строка подключения к базе данных находится в файле appsettings.json этого проекта:

- *CLI*: `dotnet ef dbcontext scaffold name=DefaultConnection Microsoft.EntityFrameworkCore.SqlServer`;
- *PMC*: `Scaffold-DbContext -Connection name=DefaultConnection -Provider Microsoft.EntityFrameworkCore.SqlServer`.

ПРИМЕЧАНИЕ Есть много команд со множеством параметров, и потребуется немало страниц, чтобы воспроизвести документацию по EF Core. Поэтому направляю вас к справочнику по командной строке EF Core на странице http://mng.bz/MXEn.

9.7.2 Установка и запуск команды обратного проектирования Power Tools

Расширение EF Core Power Tools для Visual Studio создано и поддерживается Эриком Эйлсковом Йенсеном. Его ник на GitHub и в Twitter – @ErikEJ. Этот инструмент использует службу обратного проектирования EF Core, но предоставляет визуальный интерфейс, упрощающий его использование. Это полезное расширение, потому что для кода обратного проектирования часто требуется много параметров, включая длинные строки подключения. Кроме того, данный инструмент добавляет ряд функций, например возможность настраивать шаблоны, создающие код.

Сперва необходимо установить расширение. Его можно найти на странице http://mng.bz/Gx0v. Если вы незнакомы с установкой расширений Visual Studio, см. http://mng.bz/zxBB.

После установки расширения щелкните правой кнопкой мыши по проекту в Обозревателе решений Visual Studio. Вы должны увидеть команду EF Core Power Tools с вложенной командой Reverse Engineering. Прочтите wiki-страницу EF Core Power Tools в ее репозитории на GitHub (https://github.com/ErikEJ/EFCorePowerTools/wiki).

9.7.3 Обновление классов сущности и DbContext при изменении базы данных

Один из способов обработки изменений в базе данных – обновить базу данных, а затем запустить инструмент обратного проектирования для воссоздания классов сущностей и DbContext приложения. Таким образом, вы будете уверены, что схема базы данных и модель EF Core «идут в ногу».

Инструмент обратного проектирования EF Core работает напрямую, но вы должны помнить все настройки для каждого запуска. В проекте EF Core есть функция в очереди на реализацию (задача #831), которая пытается сохранить текущий класс и изменить только свойства и связи, которые изменились. Это было бы замечательно, но такую функцию сложно реализовать, поэтому какое-то время данный вариант не рассматривался. К счастью, есть достойная замена – расширение EF Core Power Tools.

EF Core Power Tools было спроектировано с целью упростить обновление классов сущностей и DbContext приложения, предоставляя такие удобные возможности, как запоминание последнего запуска путем добавления файла в проект. Когда я разговаривал с Эриком Эйлсковом, он сказал, что использует проект базы данных SQL Server (.sqlproj), чтобы сохранить схему SQL Server в системе управления версиями, а полученные в итоге файлы SQL Server .dacpac – для обновления базы данных и EF Core Power Tools для обновления кода.

Для меня обратная сторона использования обратного проектирования при обработке миграций заключается в том, что я не могу с легкостью изменить классы сущностей, например, в соответствии со стилем предметно-ориентированного проектирования (DDD) (см. главу 13). Но можно использовать обратное проектирование один раз, чтобы получить классы сущностей и DbContext, а затем переключиться на использование кода EF Core в качестве источника истины. После этого можно отредактировать классы сущностей, чтобы получить желаемый стиль, но далее необходимо перейти на изменения базы данных через миграции EF Core или SQL-сценарии.

ПРИМЕЧАНИЕ Некоторые из моих экспериментов с обратным проектированием показывают, что параметры конфигурации EF Core `OnDelete` – не совсем такие, какие я ожидал; см. проблему EF Core #21252. Когда я спросил Эрика об этой ситуации, он ответил примерно так: настройка `ON DELETE` в базе данных верна, и это важно.

9.8 Часть 2: применение миграций к базе данных

До этого момента мы рассматривали различные способы миграции базы данных. В этом разделе мы увидим, как применить миграцию к базе данных. То, как вы создаете миграцию, влияет на то, как можно ее применить. Например, если вы создавали свои миграции, используя сценарии изменения SQL, вы не сможете применить их, используя метод EF Core `Migrate`. Вот список техник, которые мы рассмотрим в оставшейся части этой главы:

- вызов метода EF Core `Database.Migrate` из основного приложения;
- выполнение метода `Database.Migrate` из отдельного приложения, предназначенного только для миграции базы данных;
- использование сценария изменения SQL в качестве миграции EF Core и его применение к базе данных;
- применение сценариев изменения SQL с помощью инструмента миграции.

Другая проблема, которая влияет на то, как вы выполняете миграцию базы данных, – это окружение, в котором вы работаете, в частности характеристики приложения, которое обращается к обновляемой базе данных, где особое внимание нужно уделить вашей рабочей системе. Первая характеристика – запускаете ли вы несколько экземпляров приложения, например несколько экземпляров ASP.NET Core, что в Microsoft Azure называется горизонтальным масштабированием. Это важная характеристика, потому что все способы применения миграции к базе данных полагаются только на одно приложение, пытающееся изменить схему базы данных. Следовательно, наличие нескольких запущенных экземпляров исключает некоторые более простые методы обновления миграции, такие как запуск миграции при запуске приложения, потому что все экземпляры будут пытаться запуститься одновременно (но см. решение, предложенное @zejji, для этого ограничения в примечании в разделе 9.8.1).

Вторая характеристика – применяется ли миграция во время работы текущего приложения. Такая ситуация возникает, если у вас есть приложения, которые должны работать постоянно, например системы электронной почты и сайты, к которым люди хотят получать доступ в любое время, скажем GitHub и Amazon. Я называю эти типы приложений приложениями непрерывного цикла.

Любая миграция, применяемая к базе данных такого приложения, не должна нарушать его работу; база данных должна по-прежнему работать с кодом запущенного приложения. Если вы добавите столбец, не допускающий значения NULL, без значения по умолчанию, то, например, когда старое приложение создаст новую строку, база данных отклонит ее, поскольку старое приложение не предоставило значение для заполнения нового столбца. Это критическое изменение нужно разделить на серию некритических изменений, как описано в разделе 9.9.2.

В последующих разделах рассматриваются четыре способа применения миграции к базе данных на основании характеристик приложения. Некоторые самые сложные проблемы, связанные с изменениями схемы базы данных, описаны в разделе 9.9.

9.8.1 Вызов метода Database.Migrate из основного приложения

Вы видели этот подход для ASP.NET Core в разделе 5.9.2, но, резюмируя, мы добавляем некий код, который вызывает метод `context.Database.`

`Migrate` перед запуском основного приложения. Такой подход – безусловно, самый простой способ применить миграцию, но у него есть существенное ограничение: не следует делать несколько вызовов метода `Migrate` одновременно. Если у приложения несколько экземпляров, работающих параллельно, – характерная черта многих приложений, – этот подход использовать нельзя. В табл. 9.6 представлен его обзор.

Таблица 9.6 Краткий обзор достоинств и недостатков, а также ограничений вызова метода `Database.Migrate` из основного приложения

	Примечания
Положительные стороны	▪ Относительно легко реализовать. ▪ Гарантирует, что база данных будет обновлена до запуска вашего приложения
Отрицательные стороны	▪ *Нельзя* параллельно вызывать метод `Migrate` из двух и более приложений. ▪ Есть небольшой период, в течение которого приложение не отвечает (см. примечание после этой таблицы). ▪ Если в ходе миграции произошла ошибка, приложение завершит работу. ▪ Иногда бывает сложно диагностировать ошибки запуска
Ограничения	Этот подход не работает, если запущено несколько экземпляров приложения (но см. решение, предложенное @zejji, для этого ограничения после данной заметки)
Советы	Для приложений ASP.NET Core я по- прежнему рекомендую применять миграцию в конвейере непрерывной интеграции и доставки (CI/CD), даже если вы планируете запустить только один экземпляр веб-приложения (см. раздел 9.8.2), потому что в случае сбоя миграции ваше приложение не будет развернуто, и при необходимости вы будете готовы к масштабированию
Мой вердикт	Если вы можете гарантировать, что одновременно запускается только один экземпляр вашего приложения, этот подход является простым решением для миграции базы данных. К сожалению, такая ситуация нетипична для сайтов и локальных приложений

ПРИМЕЧАНИЕ Этот подход предполагает, что вы развертываете свое приложение без использования каких-либо постоянно работающих функций, таких как слоты развертывания Azure Web App и переключение. В этом случае старое приложение будет остановлено до запуска нового. В течение этого (непродолжительного) времени любой доступ к приложению окончится неудачей, что может привести к потере редактируемых данных.

Данный подход используется в приложении BookApp из связанного репозитория на GitHub. Это означает, что вы можете запустить приложение на компьютере разработчика, и база данных будет создана автоматически (если у вас установлен localdb). Это показывает, насколько он полезен. Но для приложений, которые необходимо масштабировать, такой подход не сработает.

ПРИМЕЧАНИЕ Пользователь с ником @zejji опубликовал подход, гарантирующий однократный вызов метода `Migrate` в приложении, запущенном в нескольких экземплярах. Такой подход решает одну из проблем вызова этого метода при запуске; см. http://mng.bz/VGw0.

ПОИСК МИГРАЦИЙ, КОТОРЫЕ МЕТОД DATABASE.MIGRATE ПРИМЕНИТ К БАЗЕ ДАННЫХ

При использовании метода `context.Database.Migrate` для миграции базы данных вы, возможно, захотите запустить некий код C#, если применяется определенная миграция. Я использовал эту технику для заполнения нового свойства/столбца, добавленного при определенной миграции. Можно узнать, какие миграции будут применены к базе данных, вызвав метод `GetPendingMigrations` перед вызовом методов `Migrate` и `GetAppliedMigrations`, чтобы получить миграции, которые были применены к базе данных.

Оба метода возвращают набор строк с именами файлов, содержащих миграцию.

У BookApp, например, есть класс миграции `InitialMigration`, который находится в файле с именем наподобие 20200507081623_InitialMigration. В следующем листинге показано, как определить, что класс `InitialMigration` был применен, чтобы можно было запустить код C# в перенесенной базе данных.

Листинг 9.9 Определяем миграции, примененные к базе данных

Вызываем метод миграции, чтобы применить любые отсутствующие миграции к базе данных

Используем метод расширения, чтобы узнать, применена ли InitialMigration к базе данных

```
context.Database.Migrate();
if (context.CheckIfMigrationWasApplied(nameof(InitialMigration)))
{
    //... Выполняем код C# для этой конкретной миграции;
}

// Был применен метод расширения для обнаружения конкретной миграции;
public static bool CheckIfMigrationWasApplied(
    this DbContext context, string className)
{
    return context.Database.GetAppliedMigrations()
        .Any(x => x.EndsWith(className));
}
```

Код, который должен выполняться после применения InitialMigration

Простой метод расширения для обнаружения конкретной миграции по имени класса

Метод GetAppliedMigrations возвращает имя файла для каждой миграции, примененной к базе данных

Все имена файлов заканчиваются именем класса, поэтому мы возвращаем true, если какое-либо имя файла заканчивается на className

Я использовал этот подход для получения надлежащего эффекта, но имейте в виду, что если ваш C#-код слишком долго выполняется в приложении ASP.NET Core, то ваш веб-сервер может отключить приложение, и в этом случае выполнение дополнительного кода обновления миграции C# будет остановлено посреди работы.

9.8.2 Выполнение метода Database.Migrate из отдельного приложения

Вместо того чтобы запускать миграцию как часть кода запуска, можно создать отдельное приложение, чтобы применить миграцию к базе данных. Можно добавить проект консольного приложения к своему решению, например используя DbContext приложения для вызова метода `context.Database.Migrate` при его запуске, возможно, принимая строку подключения к базе данных в качестве параметра. Еще один вариант – вызвать команду `dotnet ef database update`, которая в EF Core 5 может принимать строку подключения. Этот подход может применяться как в запущенном, так и в остановленном приложении. Данный раздел предполагает, что приложение остановлено. В табл. 9.7 представлен его обзор. Подход, который используется для работающего приложения, описывается в разделе 9.9.

Таблица 9.7 Краткое изложение плюсов, минусов и ограничений выполнения метода `Database.Migrate` из отдельного приложения

	Примечания
Положительные стороны	▪ Если миграция завершилась ошибкой, вы получите надлежащую обратную связь. ▪ Данный подход решает проблему потокобезопасности метода `Migrate`
Отрицательные стороны	▪ Приложение не работает во время применения миграции (но см. раздел 9.9, где приводится пример миграции базы данных во время работы приложения)
Ограничения	Никаких
Мой вердикт	Это удобный вариант, если у вас несколько экземпляров приложения. В конвейере CI/CD, например, можно остановить текущие приложения, запустить одну из команд EF Core `Migrate` (например, `dotnet ef database update`), а затем загрузить и запустить новое приложение

Если приложения не обращаются к базе данных, возможно, это связано с тем, что все они остановлены и нет никаких проблем, связанных с применением миграции к базе данных. Такой подход я называю *закрыто на техобслуживание*; подробности см. на рис. 9.8.

9.8.3 Применение миграции EF Core с помощью SQL-сценария

В некоторых случаях вам нужно использовать миграции EF Core, но вы хотите проверить или применить их, используя сценарии изменения SQL. Можно заставить EF Core создавать сценарии изменения, но если вы воспользуетесь этим подходом, обратите внимание на некоторые моменты.

Например, сценарий изменения SQL по умолчанию, созданный EF Core, содержит только сценарий для обновления базы данных без проверки того, была ли миграция уже применена. Причина состоит в том, что разработчики обычно применяют сценарии изменения SQL, используя некую систему развертывания, которая берет на себя

работу по определению того, какие миграции необходимо применить к базе данных. В табл. 9.8 представлен обзор этого подхода.

ПРИМЕЧАНИЕ Также есть способ вывести сценарий, который проверяет, была ли применена миграция. Об этом говорится в конце данного раздела.

Таблица 9.8 Краткое изложение преимуществ и недостатков, а также ограничений применения миграции EF Core с помощью SQL-сценариев

	Примечания
Положительные стороны	▪ EF Core создаст миграции за вас, а затем предоставит вам миграцию в виде кода SQL. ▪ Сценарии SQL, созданные EF Core, обновляют таблицу истории миграций
Отрицательные стороны	▪ Необходимо приложение, чтобы применить миграции к базам данных
Ограничения	Никаких
Советы	▪ Имейте в виду, что отдельные миграции не проверяют, была ли миграция применена к базе данных. Этот подход предполагает, что какое-то другое приложение отслеживает миграции. ▪ Если вам нужна миграция, которая проверяет, применялась ли она к базе данных, нужно добавить в команду параметр `idempotent`
Мой вердикт	Если вы хотите проверить или отменить миграцию либо использовать более комплексную систему развертывания приложений / баз данных, такую как Octopus Deploy или продукт RedGate, этот подход – правильный выбор

Основная команда для преобразования последней миграции в сценарий SQL:

- *CLI*: dotnet ef migrations script;
- *PMC*: Script-Migration.

Эти две команды выводят код SQL последней миграции, не проверяя, была ли эта миграция применена к базе данных. Но когда вы добавляете для этих команд параметр `idempotent`, код SQL, который они генерируют, содержит проверки таблицы истории миграций и применяет только те миграции, которые не были применены к базе данных.

ПРИМЕЧАНИЕ Есть ряд команд со множеством параметров, и потребуется немало страниц, чтобы воспроизвести документацию по EF Core. Поэтому посетите страницу http://mng.bz/MXEn, где приводится справка по командной строке EF Core.

Начиная с EF Core версии 5 сценарий SQL, созданный командой `Script-Migration`, применяет миграцию внутри транзакции SQL. Если ошибки не будет, то вся миграция будет применена к базе данных. В противном случае ни одно из изменений применено не будет.

ВНИМАНИЕ У SQLite имеются некоторые ограничения на применение миграции внутри транзакции, поскольку некоторые команды миграции сами используют транзакции. Это означает, что в случае сбоя миграции часть изменений может быть применена.

9.8.4 Применение сценариев изменения SQL с помощью инструмента миграций

Если вы выбрали подход, основанный на сценариях изменения SQL, то, вероятно, уже знаете, как будете применять эти сценарии к базе данных. Вам нужно будет использовать инструмент миграций, такой как DbUp (с открытым исходным кодом), либо бесплатные или коммерческие инструменты, такие как Flyway от компании RedGate. Как правило, у этих инструментов есть собственная версия таблицы истории миграций EF Core. (В DbUp эта таблица называется SchemaVersions.)

Реализация миграции зависит от используемого инструмента. DbUp – например, это пакет NuGet, поэтому можно использовать его так же, как метод `Migrate`: вызовите его при запуске или как отдельное приложение в конвейере CI/CD и т. д. Другие инструменты миграции нельзя вызывать из NET Core, но они используют некую форму командной строки или интеграцию с конвейером развертывания. В табл. 9.9 приводится обзор этого подхода.

Таблица 9.9 Обзор плюсов, минусов и ограничений применения сценариев изменения SQL с использованием инструмента миграций

	Примечания
Положительные стороны	▪ Инструмент работает в любых ситуациях. ▪ Хорошо работает с системами развертывания
Отрицательные стороны	▪ Вы должны самостоятельно управлять сценариями и следить за тем, чтобы их имена определяли порядок, в котором они будут применяться
Ограничения	Никаких
Советы	Когда я применял этот подход, то проводил модульный тест, чтобы проверить, действительно ли созданная тестовая база данных совпала с внутренней моделью EF Core с помощью EfSchemaCompare (см. раздел 9.6.3)
Мой вердикт	Я использовал сценарии изменения SQL и DbUp в нескольких клиентских проектах, и они хорошо себя зарекомендовали. После улучшений в EF Core, возможно, у меня возникнет соблазн вернуться к использованию миграции EF Core

9.9 Миграция базы данных во время работы приложения

В разделе 9.8 было дано определение двух характеристик приложения, взаимодействующего с базой данных, и одна из них заключалась в том, всегда ли приложение должно быть доступно (приложение непрерывного цикла). Миграция базы данных во время работы приложения требует дополнительной работы, о чем и пойдет речь в этом разделе.

Для начала сравним два типа приложений: то, которое можно остановить для миграции или обновления программного обеспечения,

и то, которое должно продолжать работать, пока идет обновление (рис. 9.8).

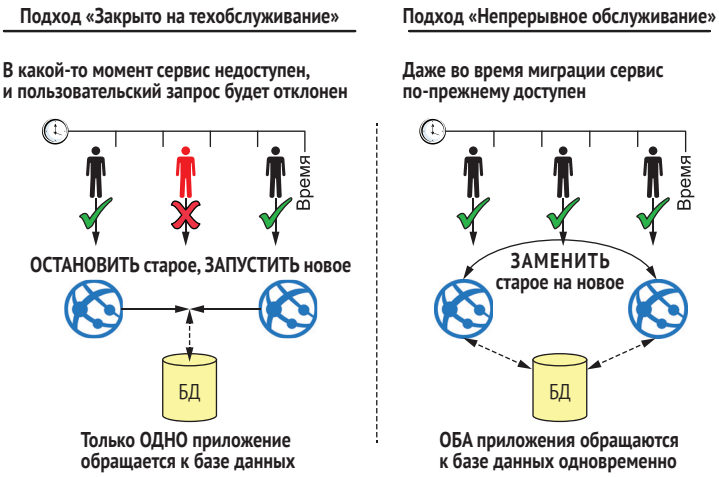


Рис. 9.8 Пример слева – то, что произойдет, если заменить старое приложение новым, – в данном случае также включая миграцию базы данных. В этом сценарии есть период времени, известный как *время простоя*, когда ни старое, ни новое приложения не работают, поэтому существует вероятность, что запрос пользователя будет потерян или отклонен. В примере справа есть существующее приложение, предоставляющее услугу, и запускается новая версия приложения, готового к работе. Когда новое приложение запускается, оно применяет миграцию к базе данных. Когда новое приложение готово, происходит «замена», и оно незаметно принимает на себя обслуживание

В оставшейся части этого раздела обсуждается, как перенести базу данных для приложения с непрерывным циклом. Возможны две ситуации:

- миграция не содержит никаких изменений, которые могли бы привести к сбою текущего работающего приложения (называемого исходным приложением);
- миграция содержит изменения, которые могут привести к сбою исходного приложения.

Что следует учитывать при остановке приложения для обновления базы данных

Нужно подумать о том, что произойдет, если резко остановить работу приложения. Это событие может привести к тому, что пользователи потеряют данные безвозвратно, или пользователь сайта онлайн-магазина может потерять заказ. По этой причине следует подумать о предупреждении или плавной остановке.

У меня была такая проблема в системе электронной коммерции, которую я создал несколько лет назад, и я разработал подход «закрыто на техобслуживание». При таком подходе на экране появляется предупреждение для пользователей, указывая на то, что сайт закроется через определенное количество минут. Во время закрытия я показывал страницу с надписью «Сайт закрыт на техническое обслуживание» и не давал пользователям доступ к другим страницам. Прочитать об этом проекте можно на странице http://mng.bz/mXkN, но имейте в виду: я создал его в 2016 г. с помощью ASP.NET MVC.

Еще один способ мягко остановить работу своего приложения – предоставить доступ к базе данных только для чтения. Вы отключаете все методы обновления базы данных. Приложение все еще читает базу данных, поэтому вы не можете изменить существующие структуры базы данных, но можете добавлять новые таблицы и безопасно копировать в них данные. После загрузки нового приложения вы можете применить еще одно обновление схемы базы данных, чтобы удалить части базы данных, которые больше не нужны.

9.9.1 Миграция, которая не содержит критических изменений

При работе над новым приложением с новой базой данных я стараюсь расширять схему базы данных по мере продвижения проекта, возможно, путем добавления новых таблиц, о которых предыдущие версии программного обеспечения не знают. Эти типы дополнений обычно не приводят к миграции, которая нарушает работу приложения, запускаемого в промышленном окружении. Приложив немного дополнительных усилий, часто можно создавать миграции, которые легко применить к приложению с непрерывным циклом. Вот некоторые вопросы, которые следует учитывать:

- если вы добавляете новое скалярное свойство в существующую таблицу, то старое приложение не установит его. Это нормально, потому что SQL предоставит ему значение по умолчанию. Но какое значение по умолчанию у свойства вам нужно? Можно управлять этим параметром, задав значение SQL по умолчанию для столбца (см. главу 10) или сделав его допускающим значение NULL. Таким образом, существующее приложение, работающее в промышленном окружении, не выйдет из строя, если оно добавит новую строку;
- если вы добавляете новый столбец внешнего ключа в существующую таблицу, необходимо сделать так, чтобы этот ключ допускал значение NULL и имел правильные настройки каскадного удаления. Такой подход позволяет старому приложению добавлять новую строку в эту таблицу без сообщения об ошибке ограничения по внешнему ключу.

СОВЕТ Настоятельно рекомендуется проводить тестирование (предположительно) некритического изменения базы данных, которое изменяет столбцы в существующих таблицах, особенно если речь идет о рабочей базе данных.

Некоторые из этих проблем, такие как создание столбца, допускающего значение NULL, когда он обычно не допускает этого значения, могут потребовать второй миграции, чтобы изменить поддержку значений NULL для столбцов базы данных, когда новое приложение заработает. Эта ситуация приводит к использованию многоэтапного подхода к миграции для работы с критическими изменениями приложения.

9.9.2 Работа с критическими изменениями, когда вы не можете остановить приложение

Применение критической миграции к приложению непрерывного цикла – одна из самых сложных миграций. Те немногие разработчики, с которыми я разговаривал, работающие над приложениями непрерывного цикла, изо всех сил стараются избегать подобного рода миграций. Как было сказано в разделе 9.9.1, некритические изменения – это норма, поэтому для (редких?) критических изменений приложения можно рассмотреть подход «закрыто на техобслуживание». Но если вам действительно нужно внести критическое изменение в приложение непрерывного цикла, читайте дальше.

В качестве примера мы рассмотрим возможность работы с миграцией базы данных, которая перемещает столбцы из таблицы «Пользователи» в новую таблицу «Адреса». В исходной миграции в разделе 9.5.2 эта проблема «перемещения столбцов» решалась одной миграцией, но она работала только потому, что работа исходного приложения была остановлена, а после завершения миграции запускалось новое приложение.

Для приложения с непрерывным циклом задача перемещения столбцов должна быть разбита на ряд этапов, чтобы ни одна миграция не нарушала работу двух приложений, работающих одновременно. В итоге мы получаем три миграции:

- `ADD` – первая миграция применяется, пока приложение App1 работает и добавляет новые функции базы данных, необходимые для запуска нового временного приложения (App2);
- `COPY` – вторая миграция применяется после остановки работы приложения App1 и до того, как было запущено целевое приложение App3. Эта миграция приводит базу данных к ее окончательной структуре;
- `SUBTRACT` – последняя миграция – это очистка, которая выполняется только тогда, когда работа приложения App2 остановлена и в дело вступает приложение App3. На этом этапе оно может

удалить старые таблицы и столбцы, которые теперь являются лишними.

Миграции `ADD`, а затем `SUBTRACT`, возможно, с `COPY` посередине, представляют общий подход к применению критических изменений в приложениях непрерывного обслуживания.

Ни при каких обстоятельствах база данных не должна быть неактуальной для двух запущенных приложений. В этом примере у нас есть пять этапов, как показано на рис. 9.9.

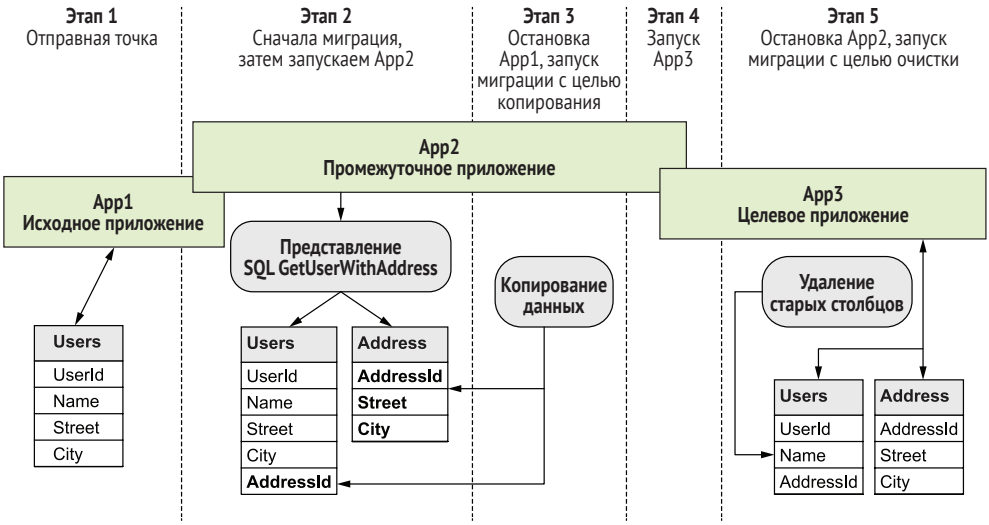


Рис. 9.9 Пять этапов, чтобы не нарушить синхронизацию базы данных, когда запущены два приложения. Первая миграция изменяет базу данных таким образом, чтобы приложение App2 могло работать с App1; следующая изменяет базу данных, так чтобы приложение App3 могло работать с App2; а финальная миграция очищает базу данных

Вот подробное описание этих этапов:

- *этап 1* – отправная точка с запущенным исходным приложением App1;
- *этап 2* – самый сложный. Он делает следующее:
  1. запускает миграцию, которая создает новую таблицу Addresses и связывает ее с текущим пользователем;
  2. добавляет представление SQL, которое возвращает пользователя с его адресом из старых столбцов Street/City или из новой таблицы Address;
  3. временное приложение, App2, использует представление SQL для чтения пользователя, но если ему необходимо добавить или обновить адрес пользователя, оно будет использовать новую таблицу Address;
- *этап 3* – работа приложения App1 остановлена, поэтому новые адреса невозможно добавить в таблицу Users. На этом этапе вы-

полняется вторая миграция, которая копирует все адресные данные из таблицы Users в новую таблицу Addresses;
- *этап 4* – на этом этапе можно запустить целевое приложение App3; оно получает адрес пользователя только из новой таблицы Addresses;
- *этап 5* – работа приложения App2 остановлена, поэтому никто не обращается к адресной части старой таблицы Users. На этом этапе выполняется последняя миграция, очищающая базу данных путем удаления столбцов Street и City из таблицы Users, а также удаляя представление SQL, необходимое для App2, и исправляя связь Пользователь/Адрес по мере необходимости.

Я мог бы привести весь код и миграции для этого примера, но в целях экономии места я сэмулировал эту многоступенчатую миграцию в модульном тесте Ch09_FiveStepsMigration, который можно найти на странице http://mng.bz/0m2N. Таким образом, вы сможете увидеть весь процесс и запустить его.

Резюме

- Самый простой способ создать миграцию – использовать миграции EF Core, но если у вас есть миграция, которая удаляет или перемещает столбцы, то нужно прибегнуть к ручному редактированию, прежде чем миграция заработает.
- Можно создавать сценарии изменения SQL, используя инструмент сравнения баз данных, или делать это вручную. Такой подход дает полный контроль над базой данных. Но нужно убедиться, что сценарии изменения создают базу данных, соответствующую внутренней модели базы данных EF Core.
- Если у вас есть существующая база данных, то можно использовать команду EF Core `scaffold` или более наглядное расширение EF Core Power Tools Visual Studio для создания классов сущностей и DbContext приложения со всеми его конфигурациями.
- Обновление рабочей базы данных – серьезное мероприятие, особенно если данные могут потеряться. То, как вы применяете миграцию к рабочей системе, зависит от типа миграции и характеристик вашего приложения.
- Есть несколько способов применить миграцию к базе данных. У самого простого подхода есть значительные ограничения, но комплексные подходы могут справиться со всеми требованиями к миграции.
- Применение миграции к базе данных во время работы приложения требует дополнительной работы, особенно если миграция изменяет схему базы данных до такой степени, что работа текущего приложения завершается ошибкой.

Для читателей, знакомых с EF6:

- миграции в EF Core были значительно изменены и улучшены, но у того, кто выполнял миграции в EF6, не должно возникнуть проблем с переходом на систему миграций EF Core;
- в EF Core нет автоматической миграции; вы контролируете процесс ее выполнения;
- в EF Core проще комбинировать миграции в команде из нескольких человек.

10 Настройка расширенных функций и разрешение конфликтов параллельного доступа

В этой главе рассматриваются следующие темы:

- использование пользовательских функций SQL в запросах EF Core;
- настройка столбцов для получения значений по умолчанию или вычисляемых значений;
- настройка свойств столбца SQL в базах данных, созданных не EF Core;
- обработка конфликтов параллельного доступа.

В этой главе обсуждается несколько дополнительных особенностей конфигурации, которые взаимодействуют напрямую с базой данных SQL, например использование *пользовательских функций* SQL и вычисляемых столбцов. Эти особенности позволяют перемещать некоторые расчеты или настройки в базу данных SQL. Хотя вы не будете использовать их ежедневно, в определенных обстоятельствах они могут быть полезны.

Вторая половина этой главы посвящена работе с несколькими, почти одновременными обновлениями одного и того же фрагмента данных в базе данных; эти обновления могут вызвать проблемы, из-

вестные как *конфликты параллельного доступа*. Вы узнаете, как настроить одно свойство/столбец или целую сущность/таблицу, чтобы перехватывать эти конфликты, а также способы их обнаружения и исправления.

10.1 DbFunction: использование пользовательских функций с EF Core

В SQL есть т. н. пользовательские функции, позволяющие писать код SQL, который будет выполняться на сервере базы данных. Эти функции полезны тем, что вы можете переместить вычисление из вашей программы в базу данных, что может быть более эффективным, поскольку вычисление может получить доступ к базе данных напрямую. Пользовательские функции, которые возвращают один-единственный результат, называются *скалярными*. А функции, которые могут возвращать набор данных, называются *табличными*. EF Core поддерживает оба типа.

> **ОПРЕДЕЛЕНИЕ** *Пользовательская функция* SQL – это процедура, которая принимает параметры, выполняет действие SQL (например, сложное вычисление) и возвращает результат этого действия в виде значения. Возвращаемое значение может быть скалярным (одиночным) значением или таблицей. Пользовательские функции отличаются от *хранимых процедур SQL* (StoredProc) тем, что пользовательские функции могут только выполнять запрос к базе данных, тогда как хранимая процедура может изменять базу данных.

Пользовательские функции особенно полезны, если вы хотите повысить производительность запроса EF Core. Я нашел код SQL (https://stackoverflow.com/a/194887/1434764), который быстрее, чем EF Core, при создании строки имен авторов, разделенных запятыми. Поэтому, вместо того чтобы преобразовывать весь запрос списка книг из приложения Book App в SQL, я могу заменить только ту часть, которая возвращает имена авторов в виде строки, разделенной запятыми. Ниже приводятся шаги по использованию пользовательских функций в EF Core.

Конфигурация:

1. Определите метод с правильным именем, входными параметрами и типом возвращаемого значения, который соответствует определению пользовательской функции. Этот метод действует как ссылка на нее.
2. Объявите метод в DbContext приложения или (что необязательно) в отдельном классе, если это скалярная функция.

3 Добавьте команды конфигурации EF Core для отображения статического метода в вызов кода пользовательской функции в базе данных.

Настройка базы данных:

4 Вручную добавьте код пользовательской функции в базу данных с помощью команды SQL.

Использование:

5 Теперь можно использовать статическую ссылку в запросе. EF Core преобразует этот метод в вызов кода пользовательской функции в базе данных.

Давайте подробнее рассмотрим все три этапа: конфигурацию, настройку базы данных и использование.

ПРИМЕЧАНИЕ Этапы конфигурации и настройки базы данных могут выполняться в любом порядке, но оба должны быть выполнены, прежде чем вы сможете использовать свою пользовательскую функцию в запросе.

10.1.1 Настройка скалярной функции

Конфигурация скалярной функции состоит из определения метода для представления пользовательской функции, а затем регистрации этого метода в EF Core во время настройки. В этом примере мы создадим пользовательскую функцию `AverageVotes`, которая вычисляет среднюю оценку в отзывах о книге. `AverageVotes` принимает первичный ключ книги, по которой нужно произвести вычисления, и возвращает значение типа double, допускающее значение NULL – `null`, если отзывов нет, или среднюю оценку, если есть отзывы.

Можно определить представление пользовательской функции как статический или нестатический метод. Нестатические методы должны быть определены в DBContext приложения; статическую версию можно определить в отдельном классе. Обычно я использую статические методы, потому что не хочу загромождать класс DBContext дополнительным кодом. На рис. 10.1 показан статический метод, который будет представлять функцию `AverageVotes` в вашей программе, а также правила формирования этого метода.

ПРИМЕЧАНИЕ Метод представления используется для определения сигнатуры пользовательской функции в базе данных: он никогда не будет вызываться как метод .NET.

Можно зарегистрировать свой статический метод пользовательской функции в EF Core, используя атрибут `DbFunction` либо Fluent API.

Вы можете использовать атрибут `DbFunction`, если разместите метод, представляющий пользовательскую функцию внутри DbContext

приложения. В примере, показанном в следующем листинге, атрибут `DbFunction` и статический метод выделены жирным шрифтом.

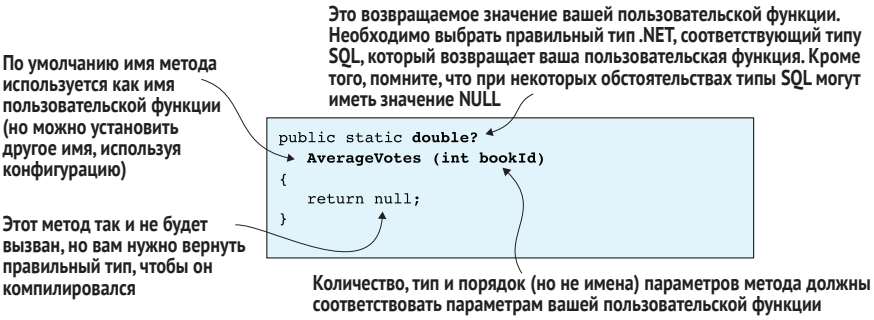


Рис. 10.1 Пример статического метода, который будет представлять вашу пользовательскую функцию внутри кода EF Core. Выноски выделяют части, которые EF Core будет использовать для отображения любых вызовов в код пользовательской функции и правила, которым вы должны следовать при создании собственного метода, который будет отображаться в пользовательскую функцию

Листинг 10.1 Использование атрибута `DbFunction` со статическим методом внутри DbContext

```
public class Chapter08EfCoreContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    //… Остальной код удален для ясности;

    public Chapter08EfCoreContext(
        DbContextOptions<Chapter08EfCoreContext> options)
        : base(options) {}

    [DbFunction]
    public static double? AverageVotes(int id)
    {
        return null;
    }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
       //… Fluent API не требуется;
    }
}
```

Атрибут DbFunction определяет метод как представление пользовательской функции

Возвращаемое значение, имя метода, а также количество, тип и порядок параметров метода должны соответствовать коду пользовательской функции

Метод не вызывается, но вам нужно вернуть правильный тип для компиляции кода

Если вы используете атрибут DbFunction, то вам не нужен Fluent API для регистрации статического метода

Другой подход – использовать Fluent API для регистрации метода как представления пользовательской функции. Преимущество такого подхода состоит в том, что вы можете разместить метод в любом классе, а это имеет смысл, если у вас много пользовательских функ-

ций. В этом листинге показан подход с использованием Fluent API для того же метода, `AverageVotes`, но он определен в классе `MyUdfMethods`, как показано на рис. 10.1.

Листинг 10.2 Регистрация статического метода, представляющего вашу пользовательскую функцию, с помощью Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    //… Остальная конфигурация удалена для ясности;

    modelBuilder.HasDbFunction(
        () => MyUdfMethods.AverageVotes(default(int)))
        .HasSchema("dbo");
}
```

Fluent API размещается в методе OnModelCreating внутри DbContext вашего приложения

HasDbFunction зарегистрирует ваш метод как способ доступа к пользовательской функции

Добавляет вызов кода пользовательской функции к вашему статическому методу

Вы можете добавить параметры. Здесь вы добавляете HasSchema (в данном случае не требуется); можно добавить HasName

После использования любого из этих подходов к настройке EF Core знает, как получить доступ к пользовательской функции в запросе.

10.1.2 Настройка табличной функции

В EF Core 5 добавлена поддержка табличных функций, что позволяет возвращать несколько значений так же, как это делает запрос к таблице. Разница заключается в том, что функция может выполнять код SQL внутри базы данных, используя параметры, которые вы ей передали.

Пример табличной функции возвращает три значения: название книги, количество отзывов и среднюю оценку. В этом примере необходимо определить класс, в экземпляре которого будут храниться три значения, возвращаемых из табличной функции, как показано в следующем фрагменте кода:

```
public class TableFunctionOutput
{
    public string Title { get; set; }
    public int ReviewsCount { get; set; }
    public double? AverageVotes { get; set; }
}
```

В отличие от скалярной функции, табличную функцию можно определить только одним способом – внутри DbContext приложения, – потому что ей нужен доступ к методу `FromExpression` класса DbContext (до выхода EF Core 5 он назывался `CreateQuery`). Вы определяете имя и сигнатуру табличной функции: имя, тип возвращаемого значения и тип параметров должны соответствовать пользовательской функции. В следующем листинге показано, как определить сигнатуру табличной функции.

Листинг 10.3 Определение табличной функции в DbContext приложения

```
public class Chapter10EfCoreContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    //… Остальной код удален для ясности;

    public Chapter10EfCoreContext(
        DbContextOptions<Chapter10EfCoreContext> options)
        : base(options) {}

    public IQueryable<TableFunctionOutput>
        GetBookTitleAndReviewsFiltered(int minReviews)
    {
        return FromExpression(() =>
            GetBookTitleAndReviewsFiltered(minReviews));
    }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {

        modelBuilder.Entity<TableFunctionOutput>()
                .HasNoKey();
        modelBuilder.HasDbFunction(() =>
            GetBookTitleAndReviewsFiltered(default(int)));
        //… Другие конфигурации опущены;
    }
}
```

Возвращаемое значение, имя метода и тип параметров должны соответствовать коду пользовательской функции

FromExpression возвращает значение типа IQueryable

Помещаем сигнатуру метода в параметр FromExpression

Мы должны настроить класс TableFunctionOutput как не имеющий первичного ключа

Регистрируем наш метод с помощью Fluent API

Может показаться странным, что мы вызываем метод внутри самого себя, но помните, что мы только определяем сигнатуру нашей пользовательской функции. EF Core заменит вызов внутреннего метода вызовом пользовательской функции, когда мы будем использовать его в запросе.

10.1.3 Добавление кода пользовательской функции в базу данных

Прежде чем использовать функцию, которую вы настроили, необходимо поместить ее код в базу данных. Обычно пользовательская функция представляет собой набор команд SQL, которые выполняются в базе данных, поэтому необходимо вручную добавить код этой функции в базу данных перед вызовом.

Первый способ – добавить ее код с помощью миграции EF Core. Для этого используется метод `migrationBuilder.Sql`, описанный в разделе 9.5.2. В главе 15 я использую две пользовательские функции для повышения производительности приложения Book App; я добавил эти функции в базу данных, отредактировав миграцию, добавив код для создания двух пользовательских функций.

Еще один подход – добавить код пользовательской функции с помощью метода EF Core `ExecuteSqlRaw` или ExecuteSqlInterpolated, описанного в разделе 11.5. Этот вариант больше подходит для модульного тестирования, где для создания базы данных миграции не используются. В этом случае нужно добавлять пользовательские функции вручную. В следующем листинге используется команда EF Core `ExecuteSqlRaw` для добавления кода SQL, определяющего функцию `AverageVotes`.

Листинг 10.4 Добавление пользовательской функции в базу данных с помощью метода `ExecuteSqlRaw`

Захватывает имя статического метода, представляющего вашу пользовательскую функцию, и использует его в качестве имени функции, добавляемой в базу данных

Использует метод EF Core ExecuteSqlRaw для добавления пользовательской функции в базу данных

Приведенный ниже код SQL добавляет пользовательскую функцию в базу данных SQL Server

```
public const string UdfAverageVotes =
    nameof(MyUdfMethods.AverageVotes);

context.Database.ExecuteSqlRaw(
    $"CREATE FUNCTION {UdfAverageVotes} (@bookId int)" +
    @" RETURNS float
     AS
     BEGIN
     DECLARE @result AS float
     SELECT @result = AVG(CAST([NumStars] AS float))
          FROM dbo.Review AS r
          WHERE @bookId = r.BookId
     RETURN @result
     END");
```

Этот код следует выполнить до того, как запросы EF Core вызовут пользовательскую функцию. Как уже было сказано, в главе 9 дается более подробная информация о том, как правильно делать это в промышленном окружении.

ПРИМЕЧАНИЕ Я не привел в этой главе SQL-код табличной функции. В репозитории на странице http://mng.bz/pJQz можно найти метод `AddUdfToDatabase`.

10.1.4 Использование зарегистрированной пользовательской функции в запросах к базе данных

Зарегистрировав пользовательские функции как отображаемые в методы представления и добавив их код в базу данных, вы готовы использовать пользовательские функции в запросах к базе данных. Ваши методы можно использовать для получения значения или как часть фильтра запроса либо сортировки. В следующем листинге приведен запрос, включающий в себя вызов скалярной функции, который возвращает информацию о книге, включая среднюю оценку по отзывам.

Листинг 10.5 Использование скалярной функции в запросе EF Core

```
var bookAndVotes = context.Books.Select(x => new Dto
{
    BookId = x.BookId,
    Title = x.Title,
    AveVotes = MyUdfMethods.AverageVotes(x.BookId)
}).ToList();
```

Обычный запрос EF Core к таблице Books

Вызывает вашу скалярную функцию, используя метод представления

Этот листинг создает следующий код SQL для выполнения в базе данных. Вызов пользовательской функции выделен жирным шрифтом:

```
SELECT [b].[BookId], [b].[Title],
[dbo].AverageVotes([b].[BookId]) AS [AveVotes]
FROM [Books] AS [b]
```

ПРИМЕЧАНИЕ EF Core может вычислять среднее значение без использования пользовательской функции с помощью команды LINQ `x.Reviews.Average(q => (double?)q.NumStars)`. Расчет средней оценки – актуальная тема в этой книге, поэтому в примере с функцией `AverageVotes` он тоже используется.

Табличная функция требует, чтобы возвращаемым значением был класс. В следующем фрагменте кода показан вызов нашей функции `GetBookTitleAndReviewsFiltered`:

```
var result = context.GetBookTitleAndReviewsFiltered(4)
    .ToList()
```

Скалярные и табличные пользовательские функции также можно использовать в любой части запроса EF Core в качестве возвращаемых значений или для сортировки и фильтрации. Вот еще один пример, в котором наша скалярная функция возвращает только книги, средняя оценка которых – 2,5 или выше:

```
var books = context.Books
    .Where(x =>
        MyUdfMethods.AverageVotes(x.BookId) >= 2.5)
    .ToList();
```

10.2 Вычисляемый столбец: динамически вычисляемое значение столбца

Еще одна полезная функция SQL – вычисляемый столбец (также известный как *генерируемый столбец*). Основная причина использования вычисляемых столбцов – переместить часть вычислений – на-

пример, конкатенации строк – в базу данных для повышения производительности. Еще один хороший вариант использования вычисляемых столбцов – возвращение полезного значения на основе других столбцов в строке. Например, вычисляемый столбец SQL, содержащий `[TotalPrice] AS (NumBook * BookPrice)`, вернет общую стоимость этого заказа, благодаря чему писать код C# станет проще.

> **EF6** Можно использовать вычисляемые столбцы в EF6.x, но EF6.x не может создавать их за вас, поэтому нужно добавить их с помощью прямой команды SQL. Сейчас EF Core предоставляет метод конфигурации для определения вычисляемых столбцов, чтобы, когда EF Core будет создавать или переносить базу данных, он добавил вычисляемый столбец.

Вычисляемый столбец – это столбец в таблице, значение которого вычисляется с использованием других столбцов в той же строке и/или встроенной функции SQL. Кроме того, можно вызывать системные или пользовательские функции (см. раздел 10.1) со столбцами в качестве параметров, что дает широкий диапазон возможностей.

Есть два типа *вычисляемых столбцов* SQL:

- вычисление выполняется каждый раз при чтении столбца. В этом разделе я называю этот тип *динамический вычисляемый столбец*;
- вычисление выполняется только при обновлении сущности. Этот тип называется *постоянным вычисляемым столбцом*, или *хранимым генерируемым столбцом*. Не все базы данных поддерживают этот тип.

В качестве примера обоих типов вычисляемых столбцов мы воспользуемся динамическим вычисляемым столбцом, чтобы получить только год рождения человека из резервного поля, содержащего дату рождения. Этот пример имитирует код из раздела 7.14.3, который скрывает точную дату рождения, но теперь код «дата -> год» выполняется в базе данных SQL.

Второй пример вычисляемых столбцов SQL – это постоянный вычисляемый столбец, который решает проблему невозможности использования лямбда-свойств в классах сущностей (см. раздел 9.4). В этом примере у нас было свойство `FullName`, которое было сформировано путем сочетания свойств `FirstName` и `LastName`, но использовать лямбда-свойство было нельзя, так как EF Core не может выполнять фильтрацию и упорядочивание, используя это свойство. Однако когда вы используете постоянный вычисляемый столбец, он вычисляется всякий раз при обновлении строки, и вы можете использовать столбец `FullName` в любой операции фильтрации, упорядочивания, поиска и других подобных операциях. Свойства в классе объявляются обычным способом, как показано в следующем листинге, но поскольку вычисляемые столбцы – это столбцы с доступом только на чтение, вы делаете метод записи закрытым.

Листинг 10.6 Класс сущности `Person` с двумя свойствами вычисляемого столбца

```
public class Person
{
    public int PersonId { get; set; }
    public int YearOfBirth { get; private set; }

    [MaxLength(50)]
    public string FirstName { get; set; }
    [MaxLength(50)]
    public string LastName { get; set; }
    [MaxLength(101)]
    public string FullName { get; private set; }

    // Остальные свойства и методы не указаны …
}
```

Поскольку вы хотите добавить индекс к FullName, необходимо сделать так, чтобы его длина и длина его частей составляла менее 450 символов

Это свойство представляет собой вычисляемый столбец. Вы задаете закрытый метод записи, так как это свойство с доступом только на чтение

Затем нужно настроить два вычисляемых столбца и индекс. Единственный способ настройки столбцов – использовать Fluent API. В этом листинге показаны различные конфигурации для класса сущности `Person`.

Листинг 10.7 Настройка двух вычисляемых столбцов, одного постоянного и индекса

```
public class PersonConfig : IEntityTypeConfiguration<Person>
{
    public void Configure
        (EntityTypeBuilder<Person> entity)
    {
        entity.Property<DateTime>("_dateOfBirth")
            .HasColumnName("DateOfBirth");

        entity.Property(p => p.YearOfBirth)
            .HasComputedColumnSql(
                "DatePart(yyyy, [DateOfBirth])");

        entity.Property(p => p.FullName)
            .HasComputedColumnSql(
                "[FirstName] + ' ' + [LastName]",
                stored:true);

        entity.HasIndex(x => x.FullName);
    }
}
```

Настраивает резервное поле с именем столбца DateOfBirth

Настраивает свойство как вычисляемый столбец и предоставляет код SQL, который будет исполнять сервер базы данных

Делает этот вычисляемый столбец постоянным

Добавляет индекс в столбец FullName, потому что вы хотите выполнять фильтрацию и сортировку по этому столбцу

На рис. 10.2 показано, что происходит при обновлении таблицы Person. EF Core знает, что таблица содержит вычисляемый столбец, поэтому она считывает значение обратно после добавления или обновления.

ПРИМЕЧАНИЕ Чтобы сосредоточиться на одном вычисляемом столбце, я показываю только значение столбца FullName, но на самом деле новые значения столбцов YearOfBirth и FullName возвращаются в экземпляр класса `Person`, потому что у класса сущности два вычисляемых столбца.

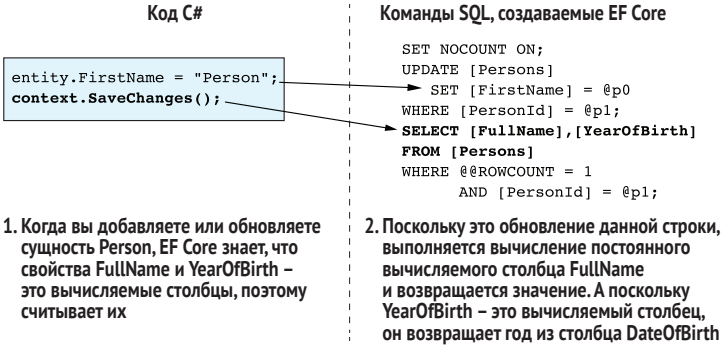


Рис. 10.2 Поскольку EF Core знает, что FullName и YearOfBirth – это вычисляемые столбцы, он считывает значения в этих двух столбцах в сущность, которая участвовала в добавлении или обновлении строки. Полное имя будет изменено, так как эта операция – обновление строки, а столбец YearOfBirth всегда пересчитывается, поэтому он также возвращается

Динамический вычисляемый столбец вычисляется повторно при каждом чтении: в случае с простыми вычислениями время вычислений будет минимальным, но если вы вызываете пользовательскую функцию, которая обращается к базе данных, то время, необходимое для чтения данных из базы, может увеличиться. Использование постоянного вычисляемого столбца преодолевает эту проблему. В некоторых типах баз данных у обоих типов вычисляемых столбцов может быть индекс, но у каждого типа базы данных есть ограничения. Например, SQL Server не позволяет индексировать вычисляемые столбцы, значение которых получено из функции даты.

10.3 Установка значения по умолчанию для столбца базы данных

Когда вы создаете экземпляр типа .NET, у него есть значение по умолчанию: 0 для `int`, `null` для string и т. д. Иногда бывает полезно задать для свойства другое значение по умолчанию. Если вы спросили кого-то, какой его любимый цвет, но он не ответил, можно было бы предоставить строку по умолчанию `not given` вместо обычного значения `null`. Можно задать значение по умолчанию в .NET с помощью инициализатора свойства C# 6.0, используя следующий код:

```
public string Answer { get; set; } = "not given";
```

Но с EF Core у вас есть два других способа установить значение по умолчанию. Во-первых, можно настроить EF Core для установки значения по умолчанию в базе данных с помощью метода Fluent API `HasDefaultValue`. Этот метод изменяет код SQL, используемый для создания таблицы в базе данных, и добавляет команду SQL `DEFAULT`, содержащую значение по умолчанию для этого столбца, если оно не указано. Как правило, это полезный подход, если строки добавляются в базу данных с помощью команд SQL, поскольку SQL часто полагается на команду `DEFAULT`, если речь идет о столбцах, для которых команда `INSERT` не предоставляет значений.

Второй подход – добавить собственный код, который будет создавать для столбца значение по умолчанию, если оно не указано. Этот подход требует, чтобы вы написали класс, наследующий от класса `ValueGenerator`, который вычислит значение по умолчанию. Затем нужно настроить свойство или свойства для использования класса `ValueGenerator` через Fluent API. Это полезный подход, когда у вас есть общий формат для определенного типа значений, например создание уникальной строки для упорядочивания книг, выбранных пользователем.

У методов установки значений по умолчанию в EF Core есть несколько общих черт. Прежде чем изучить каждый подход, давайте определим их:

- значения по умолчанию могут применяться к свойствам, резервным полям и теневым свойствам. Мы будем использовать универсальный термин *столбец*, чтобы охватить все три типа, потому что все они в конечном итоге применяются к столбцу в базе данных;
- значения по умолчанию (`int`, `string`, `DateTime`, `GUID` и т. д.) применяются только к скалярным (нереляционным) столбцам;
- EF Core предоставит значение по умолчанию, только если свойство содержит значение по умолчанию в общеязыковой среде выполнения (CLR), соответствующее его типу. Например, если для свойства типа `int` не задано значение, то в качестве значения по умолчанию будет использоваться 0;
- методы значений по умолчанию в EF Core работают на уровне экземпляра сущности, а не на уровне класса. Значения по умолчанию не будут применены, пока вы не вызовете метод `SaveChanges` или (в случае генератора значений) пока не используете команду `Add` для добавления сущности.

Для ясности: значения по умолчанию применяются только для новых строк, добавленных в базу данных, а не для обновлений. Можно настроить EF Core для добавления значения по умолчанию тремя способами:

- используя метод `HasDefaultValue` для добавления постоянного значения для столбца;
- используя метод `HasDefaultValueSql` для добавления команды SQL для столбца;

- используя метод `HasValueGenerator` для назначения свойству генератора значений.

EF6 Эти три метода установки значения по умолчанию используются в EF Core впервые. В EF6.x нет эквивалентных методов.

10.3.1 Использование метода HasDefaultValue для добавления постоянного значения для столбца

Следуя первому подходу, EF Core добавляет команду SQL `DEFAULT` для столбца, когда создает миграцию базы данных, предоставляя простую константу для установки в столбце, если создается новая строка, и свойство, отображаемое в этот столбец, имеет значение по умолчанию. Добавить команду `DEFAULT` для столбца можно только с помощью метода Fluent API `HasDefaultValue`. Следующий код устанавливает дату по умолчанию 1 января 2000 года в столбце DateOfBirth в таблице People.

Листинг 10.8 Настройка свойства для установки значения по умолчанию в базе данных SQL

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<DefaultTest>()
        .Property("DateOfBirth")
        .HasDefaultValue(new DateTime(2000,1,1));
    //… Остальные конфигурации не указаны;
}
```

Вы должны настроить установку значения по умолчанию с помощью команд Fluent API

Добавляем команду SQL DEFAULT в столбец с помощью метода HasDefaultValue

SQL-код, создаваемый EF Core для создания или миграции базы данных SQL Server, будет выглядеть, как показано ниже. Команда, устанавливающая значение по умолчанию, выделена жирным шрифтом:

```
CREATE TABLE [Defaults] (
    [Id] int NOT NULL IDENTITY,
    -- остальные столбцы пропущены
    [DateOfBirth] datetime2 NOT NULL
            DEFAULT '2000-01-01T00:00:00.000',
    CONSTRAINT [PK_Defaults] PRIMARY KEY ([Id])
);
```

Если столбец в новой сущности имеет значение по умолчанию в CLR, EF Core не предоставляет значение для этого столбца в команде `INSERT`. Это означает, что сервер базы данных применит ограничение по умолчанию для определения столбца, чтобы предоставить значение для вставки в новую строку.

ПРИМЕЧАНИЕ Если вы работаете с базой данных, созданной не EF Core, вам все равно необходимо зарегистрировать конфигурацию, потому что EF Core не должен устанавливать значение этого столбца, если значение в связанном свойстве содержит значение по умолчанию в CLR для этого типа.

10.3.2 Использование метода HasDefaultValueSql для добавления команды SQL для столбца

Установка значения по умолчанию на уровне базы данных не дает весомых преимуществ по сравнению с установкой значения по умолчанию в коде, если только ваше или другое приложение не использует прямые команды SQL для создания новой строки. Полезнее получить доступ к некоторым системным функциям SQL, которые возвращают текущую дату и время. Сделать это позволяет метод `HasDefaultValueSql`.

В некоторых ситуациях полезно узнать время добавления строки в базу данных. В таком случае, вместо того чтобы указывать константу в команде `DEFAULT`, можно предоставить функцию SQL, которая даст динамическое значение при добавлении строки в базу данных. У SQL Server, например, есть две функции – `getdate` и `getutcdate`, – которые предоставляют текущее локальное время и время в UTC соответственно. Можно использовать эти функции для автоматического определения точного времени, когда была вставлена строка. Конфигурация столбца такая же, как и у примера с константой в листинге 10.8, за исключением того, что используемая строка вызывает функцию SQL `getutcdate`, как показано в этом фрагменте кода:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<DefaultTest>()
        .Property(x => x.CreatedOn)
        .HasDefaultValueSql("getutcdate()");
    …
}
```

Если вы хотите использовать этот столбец, чтобы отслеживать, когда была добавлена строка, необходимо убедиться, что свойство .NET не задано через код (т. е. остается значением по умолчанию). Для этого используется свойство с закрытым методом записи. В следующем фрагменте кода показано свойство с закрытым методом записи, где создается простое значение отслеживания, которое автоматически сообщает, когда строка была впервые добавлена в базу данных:

```
public DateTime CreatedOn {get; private set;}
```

Это полезная возможность. Помимо доступа к системным функциям, таким как `getutcdate`, можно поместить собственные пользова-

тельские функции SQL в ограничение по умолчанию. Для команд SQL есть предел, например нельзя ссылаться на другой столбец в ограничении по умолчанию, но метод `HasDefaultValueSql` может предоставить полезные функции по сравнению с установкой значения по умолчанию в вашем коде.

10.3.3 Использование метода HasValueGenerator для назначения генератора значений свойству

Третий подход к добавлению значения по умолчанию выполняется не в базе данных, а внутри кода EF Core. EF Core разрешает, чтобы класс, наследуемый от класса `ValueGenerator` или `ValueGenerator<T>`, был настроен в качестве генератора значений для свойства или резервного поля. У этого класса будет запрошено значение по умолчанию, если оба следующих утверждения истинны:

- для свойства `State` сущности установлено значение `Added`; сущность считается новой сущностью, которая будет добавлена в базу данных;
- значение для свойства еще не задано; его значение равно значению по умолчанию для типа .NET.

В EF Core есть генератор значений, который, например, предоставит уникальные значения GUID для первичных ключей. Но для нашего примера в следующем листинге показан простой генератор значений, который создает уникальную строку, используя свойство `Name` в сущности, текущую дату в виде строки и уникальную строку из GUID для создания значения свойства `OrderId`.

Листинг 10.9 Генератор значений, который создает уникальную строку для OrderId

```
public class OrderIdValueGenerator
    : ValueGenerator<string>
{
    public override bool
        GeneratesTemporaryValues => false;

    public override string Next
        (EntityEntry entry)
        {
            var name = entry.
                Property(nameof(DefaultTest.Name))
                    .CurrentValue;
            var ticks = DateTime.UtcNow.ToString("s");
            var guidString = Guid.NewGuid().ToString();
```

Генератор значений должен наследовать от ValueGenerator<T>

Установите значение false, если хотите, чтобы ваше значение было записано в базу данных

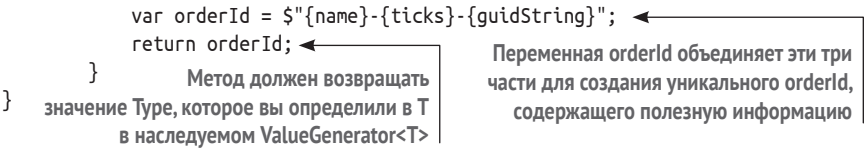
Этот метод вызывается при добавлении сущности в DbContext

Выбирает свойство Name и получает его текущее значение

Параметр дает вам доступ к сущности, для которой генератор создает значение. Вы можете получить доступ к его свойствам

Предоставляет уникальную строку

Предоставляет дату в удобном для сортировки формате



Следующий код настраивает использование генератора значений:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<DefaultTest>()
        .Property(p => p.OrderId)
        .HasValueGenerator((p, e) =>
            new OrderIdValueGenerator());
    …
}
```

Обратите внимание, что метод генератора значений `Next` вызывается, когда вы добавляете сущность через `context.Add(newEntity)`, но до того, как данные будут записаны в базу данных. При вызове метода `Next` никакие значения, предоставленные базой данных, например первичный ключ, использующие команду SQL `IDENTITY`, заданы не будут.

ПРИМЕЧАНИЕ Можно использовать метод `NextAsync`, если требуется реализовать асинхронную версию, например асинхронный доступ к базе данных при генерировании значения по умолчанию. В этом случае нужно использовать метод `AddAsync` при добавлении сущности в базу данных.

Генератор значений – это специализированная функция с ограниченным применением, но о ней стоит знать. В главе 11 показано, как перехватывать записи в базу данных, чтобы добавить отслеживание или другую информацию. На это требуется больше работы, но дает больше возможностей, чем генератор значений.

10.4 Последовательности: предоставление чисел в строгом порядке

Последовательности в базе данных позволяют создавать числа в строгом порядке без разрывов, например 1,2,3,4. Нет гарантии, что значения ключей, созданные командой SQL `IDENTITY`, будут последовательными; они могут выглядеть так: 1,2,10,11. Последовательности полезны, когда требуется гарантированная известная последовательность, например для номера заказа при совершении покупки.

Способы реализации последовательностей различаются в зависимости от сервера базы данных, но, как правило, последовательность назначается не конкретной таблице или столбцу, а схеме. Каждый раз,

когда столбцу требуется значение из последовательности, он запрашивает это значение. EF Core может настроить последовательность, а затем с помощью метода `HasDefaultValueSql` установить для столбца следующее значение в последовательности.

В следующем листинге показан класс сущности `Order` с `OrderNo`, который использует последовательность. Фрагмент с методом `HasDefaultValueSql` предназначен для базы данных SQL Server и для других серверов баз данных будет выглядеть иначе. Этот пример добавляет последовательность SQL к миграции или в базу данных, созданную с помощью метода `context.Database.EnsureCreated()`, и получает следующее значение в последовательности, задав значение по умолчанию для столбца OrderNo.

Листинг 10.10 DbContext с конфигурацией Fluent API и классом `Order`

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating
       (ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>(
                "OrderNumbers", "shared")
            .StartsAt(1000)
            .IncrementsBy(5);

        modelBuilder.Entity<Order>()
            .Property(o => o.OrderNo)
            .HasDefaultValueSql(
                "NEXT VALUE FOR shared.OrderNumbers");
   }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
}
```

Создает последовательность SQL OrderNumber в схеме «shared». Если схема не указана – используется схема по умолчанию

(Необязательно) Позволяет управлять началом и инкрементом последовательности. По умолчанию начинается с 1 и увеличивается на 1

Столбец может получить доступ к номеру последовательности через ограничение по умолчанию. Каждый раз, когда вызывается команда NEXT VALUE, последовательность увеличивается

EF6 Это нововведение в EF Core, и у него нет соответствующего аналога в EF6.

10.5 Помечаем свойства, созданные базой данных

При работе с существующей базой данных вам может потребоваться сообщить EF Core о конкретных столбцах, которые обрабатываются

иначе, чем ожидает EF Core. Если у существующей базы данных есть вычисляемый столбец, который вы не настраивали с помощью Fluent API (см. раздел 10.2), необходимо сообщить EF Core, что столбец вычисляется таким образом, чтобы его можно было обработать правильно.

Сразу скажу, что такая маркировка столбцов не является нормой, потому что EF может самостоятельно определять атрибуты столбцов на основе предоставленных вами команд конфигурации. Вам *не* нужны никакие функции из этого раздела, если вы используете EF Core для того, чтобы:

- создать или мигрировать базу данных, используя EF Core;
- выполнить обратное проектирование базы данных, как описано в главе 9.

Можно использовать эти возможности, если вы хотите применять EF Core с существующей базой данных без обратного проектирования. В этом случае нужно сообщить EF Core о столбцах, которые не соответствуют обычным правилам. В следующих разделах вы узнаете, как пометить три разных типа столбцов:

- столбцы, которые изменяются при вставке новой строки или обновлении строки;
- столбцы, которые меняются при вставке новой строки;
- «обычные» – столбцы, которые изменяются только в EF Core.

EF6 В EF6 есть аналогичные аннотации данных для настройки сгенерированных базой данных свойств, но EF Core также предоставляет версии с Fluent API.

10.5.1 Помечаем столбец, создаваемый при добавлении или обновлении

EF Core необходимо знать, генерируется ли значение столбца базой данных, как, например, в случае с вычисляемыми столбцами, хотя бы по той причине, что это столбец с доступом только на чтение. EF Core не умеет «угадывать», что именно база данных устанавливает значение столбца, поэтому нужно пометить его соответствующим образом. Можно использовать Data Annotations или Fluent API.

Аннотация данных для столбца добавления или обновления показана в следующем фрагменте кода. Здесь EF Core использует существующий параметр `DatabaseGeneratedOption.Computed`. Параметр называется `Computed`, потому что это наиболее вероятная причина, по которой столбец будет изменен при добавлении или обновлении:

```
public class PersonWithAddUpdateAttibutes
{
    …

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
```

```
    public int YearOfBirth { get; set; }
}
```

Этот фрагмент кода использует Fluent API для установки параметра добавления или обновления для столбца:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(p => p.YearOfBirth)
        .ValueGeneratedOnAddOrUpdate();
    …
}
```

10.5.2 Помечаем значение столбца как установленное при вставке новой строки

Можно сообщить EF Core, что столбец в базе данных будет получать значение через базу данных всякий раз, когда в базу данных вставляется новая строка. Две распространенные ситуации:

- с помощью команды SQL `DEFAULT`, которая предоставляет значение по умолчанию, если значение не задано в команде `INSERT`;
- с помощью некой формы генерации ключей, где основной метод – это команда SQL `IDENTITY`. В этих случаях база данных создает уникальное значение для размещения в столбце при вставке новой строки.

Если в столбце есть команда `DEFAULT`, она установит значение, если EF Core создает новую строку и значение не было предоставлено. В этом случае EF Core должен считать значение, заданное командой `DEFAULT` для столбца; в противном случае данные внутри вашего класса сущности не будут соответствовать базе данных.

Другая ситуация, при которой EF Core необходимо прочитать значение столбца, – это столбец первичного ключа, когда база данных предоставляет значение ключа, потому что EF Core не будет знать, что ключ был сгенерирован командой SQL `IDENTITY`. Эта ситуация, скорее всего, является причиной, по которой используется `DatabaseGeneratedOption.Identity`, как показано в следующем фрагменте кода:

```
public class MyClass
{
    public int MyClassId { get; set;}
    …
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int SecondaryKey { get; set;}
}
```

Второй пример делает то же самое, но использует Fluent API. Здесь у нас есть столбец с ограничением по умолчанию. Следующий фрагмент кода устанавливает это ограничение:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property("DateOfBirth")
        .ValueGeneratedOnAdd();

    …
}
```

10.5.3 Помечаем столбец/свойство как «обычное»

Все скалярные свойства, которые не являются ключами, не имеют значения SQL по умолчанию и не являются вычисляемыми столбцами, представляют собой *обычные* свойства, т. е. только вы устанавливаете значение свойства. В редких случаях вам, возможно, понадобится установить свойство как обычное, и EF Core предоставляет способы сделать это. Единственный случай, при котором этот подход может быть полезен, – это первичный ключ, использующий GUID; в этом случае ваша программа предоставляет значение.

> **ОПРЕДЕЛЕНИЕ** GUID – это *глобальный уникальный идентификатор*, 128-битное целое число, которое можно без опаски использовать где угодно. В некоторых случаях это хорошее значение для ключа. Возможно, программному обеспечению нужно определить ключ, обычно потому, что какой-то другой части ПО нужен ключ перед вставкой строки. Либо у вас есть реплицированные базы данных со вставками в обе или все базы данных, что усложняет создание уникального ключа.

Мои тесты показывают, что если использовать GUID в качестве первичного ключа, то EF Core автоматически создаст значение GUID, если оно не указано (EF Core предоставляет генератор значений для первичных ключей GUID). Кроме того, если поставщик базы данных предназначен для SQL Server, EF Core использует генератор значений `SequentialGuidValueGenerator`, который оптимизирован для использования в кластеризованных ключах и индексах сервера Microsoft SQL. Можно включить этот генератор значений, используя аннотацию данных:

```
public class MyClass
{
   [DatabaseGenerated(DatabaseGeneratedOption.None)]
   public Guid MyClassId { get; set;}
   …
}
```

Кроме того, можно использовать следующую конфигурацию Fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
```

```
    modelBuilder.Entity<MyClass>()
        .Property("MyClassId")
        .ValueGeneratedNever();
    …
}
```

10.6 Одновременные обновления: конфликты параллельного доступа

Конфликты параллельного доступа – большая тема, поэтому позвольте мне начать с объяснения того, как выглядят одновременные обновления, прежде чем объяснить, почему они могут стать проблемой и как с ними справиться. На рис. 10.3 показан пример одновременного обновления для столбца PublishedOn в базе данных. Это обновление происходит из-за двух отдельных фрагментов кода, выполняющихся параллельно, которые читают столбец, а затем обновляют его.

По умолчанию EF Core использует паттерн «Оптимистичный параллелизм». На рис. 10.3 первое обновление теряется, потому что перезаписывается вторым. Хотя часто такая ситуация является допустимой, в некоторых случаях перезапись чужого обновления представляет собой проблему. В следующих разделах объясняются недопустимые операции перезаписи, известные как *конфликты параллельного доступа,* и показано, как EF Core позволяет обнаруживать и устранять такие конфликты.

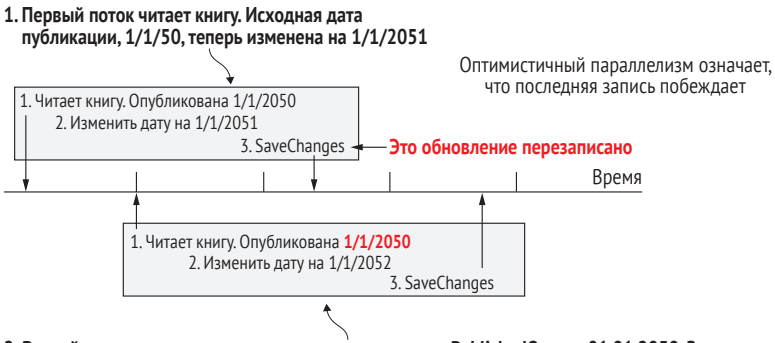


Рис. 10.3 Два фрагмента кода (скажем, в веб-приложении), выполняющиеся параллельно, которые производят почти одновременное обновление одного и того же столбца (в данном случае это дата публикации одной и той же книги). По умолчанию EF Core позволяет второй записи одержать верх, а первая запись теряется. Такая ситуация называется оптимистическим параллелизмом, но правило «побеждает последняя запись» может оказаться полезным не во всех случаях

10.6.1 Почему конфликты параллельного доступа так важны?

Если задуматься, то значение все равно можно перезаписать. Можно установить для даты публикации книги значение 01.01.2020, а завтра можно изменить его на 01.01.2040. Так почему же конфликты параллельного доступа так важны?

В некоторых случаях такие конфликты имеют значение. Например, в финансовых транзакциях можно представить, что чистота и аудит данных будут важны, поэтому может потребоваться защита от параллельных изменений. Еще один конфликт наблюдается в примере из раздела 8.7, где мы считали среднюю оценку в отзывах на книгу. В этом случае если два человека добавили отзывы одновременно, такой пересчет будет некорректным, поэтому нужно обнаружить и исправить этот конфликт, если мы хотим, чтобы данный пример надежно работал.

Могут возникать и другие конфликты такого рода на человеческом уровне. Вместо столкновения двух задач при обновлении два пользователя, которые смотрят на экран, могут столкнуться с таким же результатом: второй пользователь, щелкнувший по кнопке «Отправить», перезапишет обновление, которое, по мнению первого пользователя, было выполнено им. (Подробности описаны в разделе 10.6.4.)

Иногда конфликты параллельного доступа можно обойти на этапе проектирования, создавая приложения таким образом, чтобы одновременных опасных обновлений не было. Например, в случае с сайтом онлайн-магазина, который я разрабатывал, у меня была система обработки заказов, где использовались фоновые задачи, которые могли приводить к конфликтам параллельного доступа. Я обошел эту потенциальную проблему, спроектировав обработку заказов, чтобы исключить возможность параллельных обновлений:

- я разделил информацию о заказе клиента, выделив часть, которая никогда не меняется. Она содержала данные о том, что было заказано и куда это нужно отправить. После того как заказ создавался, они не изменялись и не удалялись;
- для изменяющейся части заказа, которая представляла собой статус заказа при его перемещении по системе, я создал отдельную таблицу, в которую добавлял каждый новый статус заказа по мере его возникновения, с указанием даты и времени. (Этот подход известен как *источники событий*.) Затем я мог получить последний статус заказа, отсортировав статусы по дате/времени и выбрав статус с самой новой датой и временем. Конечно, результат был бы неактуальным, если бы после того, как я прочитал его, был добавлен другой статус, но обработка параллельных изменений это добавление бы обнаружила.

Такой подход к проектированию означал, что я никогда не обновлял и не удалял данные о заказах, поэтому конфликтов параллельно-

го доступа просто не могло быть. Это немного усложняло обработку изменения заказа клиентом, но заказы были защищены от проблем, связанных с параллельным доступом.

Однако когда такие проблемы все же возникают и вы не можете их решить, EF Core предоставляет два способа обнаружения одновременного обновления, а при его обнаружении – способ получить все необходимые данные, чтобы вы могли написать код для устранения проблемы.

10.6.2 Возможности решения конфликтов параллельного доступа в EF Core

Функции обработки конфликтов параллельного доступа в EF Core могут обнаруживать параллельное обновление двумя способами, которые активируются добавлением к классу одной из нижеперечисленных сущностей:

- *маркера параллелизма,* чтобы пометить определенное свойство или столбец в классе сущности для проверки на предмет наличия конфликта параллельного доступа;
- *временной метки* (также известной как rowversion), которая помечает весь класс сущности или строку для проверки на предмет наличия конфликта параллельного доступа.

EF6 Функции обработки параллелизма такие же, как и в EF6.x, но в EF Core они были переписаны.

В обоих случаях при вызове метода `SaveChanges` EF Core создает код сервера базы данных для проверки на предмет наличия обновления любых сущностей, которые содержат маркеры параллелизма или временные метки. Если этот код обнаруживает, что маркеры параллелизма или временные метки изменились с тех пор, как он прочитал сущность, выбрасывается исключение `DbUpdateConcurrencyException`. В таком случае можно использовать возможности EF Core для проверки различных версий данных и использовать собственный код, чтобы решить, какому из одновременных обновлений нужно отдать предпочтение. Далее вы узнаете, как настроить оба подхода – маркер параллелизма и временную метку – и как EF Core обнаруживает изменение.

ОБНАРУЖЕНИЕ ПАРАЛЛЕЛЬНОГО ИЗМЕНЕНИЯ С ИСПОЛЬЗОВАНИЕМ МАРКЕРА ПАРАЛЛЕЛЛИЗМА

Подход с использованием маркера параллелизма позволяет настроить одно или несколько свойств в качестве маркеров параллелизма. Этот подход предписывает EF Core проверить, совпадает ли текущее значение в базе данных со значением, полученным, когда отслеживаемая сущность была загружена как часть команды SQL `UPDATE`, от-

правленной в базу данных. Таким образом, если загруженное и текущее значения из базы данных отличаются, обновление завершится ошибкой. На рис. 10.4 показан пример, где свойство `PublishedOn` помечается как маркер параллелизма, после чего возникает конфликт параллельного доступа.

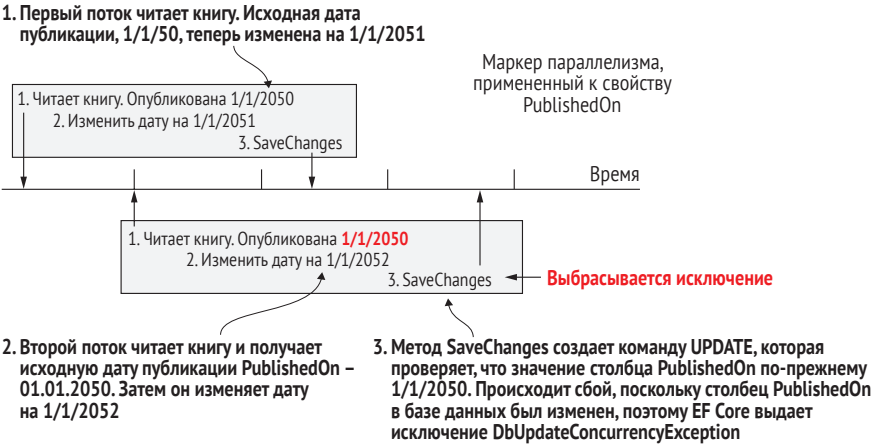


Рис. 10.4 Два фрагмента кода – скажем, в веб-приложении – выполняются параллельно, что обеспечивает почти одновременное обновление столбца `PublishedOn`. Поскольку мы пометили свойство `PublishedOn` как маркер параллелизма, EF Core использует измененную команду SQL `UPDATE`, которая выполняет обновление только в том случае, если столбец базы данных PublishedOn такой же, как и при чтении сущности `Book`. В противном случае команда `UPDATE` завершается ошибкой, а метод `SaveChanges` сгенерирует исключение `DbUpdateConcurrencyException`

Чтобы настроить этот пример, мы добавляем аннотацию данных `ConcurrencyCheck` к свойству `PublishedOn` в нашем классе сущности `ConcurrencyBook`, как показано в следующем листинге. EF Core находит эту аннотацию во время настройки и помечает свойство как маркер параллелизма.

Листинг 10.11 Класс сущности `ConcurrencyBook` со свойством `PublishedOn`

```
public class ConcurrencyBook
{
    public int ConcurrencyBookId { get; set; }
    public string Title { get; set; }

    [ConcurrencyCheck]
    public DateTime PublishedOn { get; set; }

    public ConcurrencyAuthor Author { get; set; }
}
```

Сообщает EF Core, что свойство PublishedOn – это маркер параллелизма. Это означает, что EF Core проверит, изменилось ли оно, при обновлении

В этом случае мы использовали аннотацию данных `ConcurrencyCheck` для определения свойства как маркера параллелизма. Это дает понять всем, кто смотрит на этот код, что свойство `PublishedOn` обрабатывается по-особому. В качестве альтернативы можно определить маркер параллелизма, используя Fluent API, как показано в следующем листинге.

Листинг 10.12 Установка свойства в качестве маркера параллелизма с помощью Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<ConcurrencyBook>()
        .Property(p => p.PublishedOn)
        .IsConcurrencyToken();

    //… Остальная конфигурация удалена;
}
```

Метод OnModelCreating – это то место, где размещается конфигурация обнаружения параллелизма

Определяет свойство PublishedOn как маркер параллелизма. Это означает, что EF Core проверяет, изменилось ли оно, при записи обновления

На рис. 10.4 показано, как после добавления кода из листинга 10.11 или 10.12 при вызове метода `SaveChanges` вместо перезаписи первого обновления обнаруживается, что другая задача обновила столбец PublishedOn и выбрасывается исключение.

В листинге 10.13 параллельное обновление имитируется с помощью выполнения команды SQL, которая изменяет столбец PublishedOn между блоками кода EF Core, первый из которых читает, а второй обновляет книгу. Команда SQL представляет другой поток веб-приложения или другое приложение, у которого есть доступ к той же базе данных, и возможность обновить столбец PublishedOn. В этом случае выбрасывается исключение `DbUpdateConcurrencyException`, когда в последней строке вызывается метод `SaveChanges`.

Листинг 10.13 Имитация параллельного обновления столбца PublishedOn

```
var firstBook = context.Books.First();

context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Books SET PublishedOn = GETDATE()"+
    "WHERE ConcurrencyBookId = @p0",
    firstBook.ConcurrencyBookId);
firstBook.Title = Guid.NewGuid().ToString();
context.SaveChanges();
```

Загружает первую книгу из базы данных как отслеживаемую сущность

Имитирует другой поток или приложение, изменяя столбец PublishedOn той же книги

Изменяет заголовок в книге, чтобы EF Core обновил книгу

Этот метод SaveChanges выбрасывает исключение DbUpdateConcurrencyException

Важно отметить, что проверяется только свойство, помеченное как маркер параллелизма. Если смоделированное обновление изменило,

скажем, свойство `Title`, которое не помечено как маркер параллелизма, то исключение выброшено не будет.

Этот эффект можно увидеть в коде SQL, который EF Core создает для обновления заголовка в следующем листинге. Оператор `WHERE` содержит не только первичный ключ обновляемой книги, но и столбец PublishedOn.

Листинг 10.14 Код SQL для обновления Book, где PublishedOn – маркер параллелизма

```
SET NOCOUNT ON;
UPDATE [Books] SET [Title] = @p0
WHERE [ConcurrencyBookId] = @p1
      AND [PublishedOn] = @p2;
SELECT @@ROWCOUNT;
```

Тест не проходит, если столбец PublishedOn изменился. Это останавливает обновление

Возвращает количество строк, обновленных этой командой SQL

Когда EF Core выполняет эту команду SQL, оператор `WHERE` находит строку для обновления, только если в столбце PublishedOn не изменилось значение, полученное EF Core из базы данных. Затем EF Core проверяет количество строк, обновленных командой SQL. Если количество этих строк равно нулю, EF Core выбрасывает исключение `DbUpdateConcurrencyException`, чтобы указать на наличие конфликта параллельного доступа; он может перехватить конфликт, вызванный другой задачей из-за изменения столбца PublishedOn или удаления строки, когда эта задача выполняет обновление.

Преимущество использования маркера параллелизма заключается в том, что он работает с любой базой данных, потому что использует базовые команды. Следующий способ обнаружения параллельных изменений полагается на серверную функцию базы данных.

ОБНАРУЖЕНИЕ ПАРАЛЛЕЛЬНОГО ИЗМЕНЕНИЯ ЧЕРЕЗ ВРЕМЕННУЮ МЕТКУ

Второй способ выполнить проверку на предмет наличия конфликтов параллельного доступа – использовать то, что в EF Core называется временной меткой. Временная метка работает иначе, чем маркер параллелизма, поскольку использует уникальное значение, предоставляемое сервером базы данных, которое изменяется всякий раз при вставке или обновлении строки. От параллельных изменений защищена вся сущность, а не отдельные свойства или столбцы.

На рис. 10.5 показано, что, когда строка со свойством или столбцом, помеченным как временная метка, вставляется или обновляется, сервер базы данных создает новое уникальное значение для этого столбца. Это дает эффект обнаружения обновления сущности или строки всякий раз при вызове метода `SaveChanges`.

Тип данных временной метки (timestamp) зависит от типа базы данных: в SQL Server это `ROWVERSION`, который в .NET отображается

в byte[]; в PostgreSQL есть столбец xmin – это 32-битное число без знака.

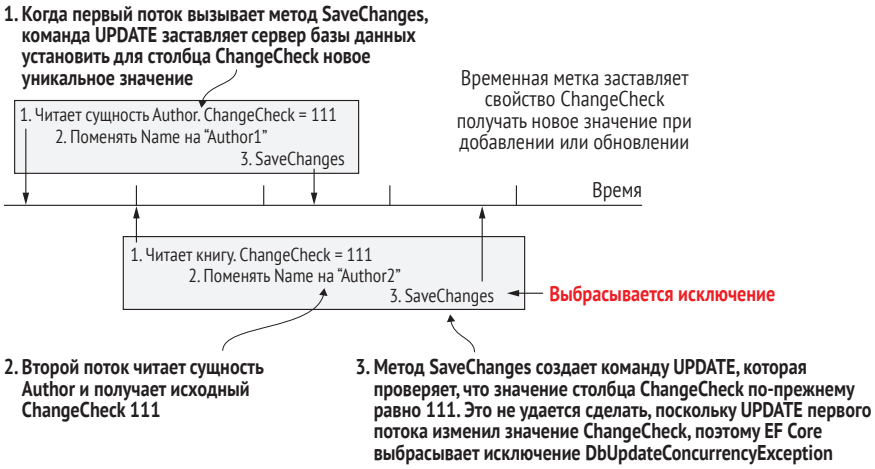


Рис. 10.5 Настройка свойства в качестве временной метки означает, что для соответствующего столбца в таблице должен быть задан тип данных, для которого будет устанавливаться новое уникальное значение каждый раз, когда к строке применяется команда SQL INSERT или UPDATE. (Если вы используете EF Core для создания своей базы данных, то поставщик базы данных обеспечит использование правильного типа столбца.) Затем, когда EF Core выполняет обновление, он проверяет, что столбец временной метки имеет то же значение, что и при чтении сущности. Если значение другое, то EF Core выбрасывает исключение

А в Cosmos DB есть JSON свойство _etag. Это строка, содержащая уникальное значение. EF Core может использовать любой из этих типов через соответствующего поставщика базы данных. В качестве примера я буду использовать временную метку SQL Server; другие базы данных будут работать аналогичным образом, но с иным типом .NET.

В следующем листинге свойство ChangeCheck, которое отслеживает любые обновления для всей сущности, добавляется в класс сущности ConcurrencyAuthor. В этом случае у свойства ChangeCheck есть аннотация данных Timestamp, сообщающая EF Core о необходимости пометить его как специальный столбец, который база данных обновит уникальным значением. В случае с SQL Server поставщик базы данных установит для столбца значение rowversion; у других баз данных иные подходы к реализации столбца TimeStamp.

Листинг 10.15 Класс ConcurrencyAuthor со свойством ChangeCheck

```
public class ConcurrencyAuthor
{
    public int ConcurrencyAuthorId { get; set; }
    public string Name { get; set; }
```

```
    [Timestamp]
    public byte[] ChangeCheck { get; set; }
}
```

Помечает свойство ChangeCheck как временную метку, вынуждая сервер базы данных пометить его как ROWVERSION. EF Core проверяет это свойство при обновлении, чтобы узнать, не изменилось ли оно

Опять же, мы используем аннотацию данных `Timestamp`, чтобы пометить свойство `ChangeCheck` как временную метку. Этот подход – рекомендуемый мною способ настройки обработки параллелизма, потому что для всех, кто смотрит на данный код, становится очевидным, что для этой сущности есть специальная обработка параллелизма. В качестве альтернативы можно использовать Fluent API для настройки временной метки, как показано в следующем листинге.

Листинг 10.16 Настройка временной метки с помощью Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<ConcurrencyAuthor>()
        .Property(p => p.ChangeCheck)
        .IsRowVersion();
}
```

OnModelCreating – это место, где вы размещаете конфигурацию обнаружения параллелизма

Определяет дополнительное свойство ChangeCheck, которое будет изменяться каждый раз при создании или обновлении строки. EF Core проверяет, изменилось ли это свойство, при обновлении

Обе конфигурации создают столбец в таблице, который сервер базы данных изменит автоматически при выполнении команды `INSERT` или `UPDATE` в этой таблице. Для базы данных SQL Server тип столбца установлен как `ROWVERSION`, как показано в следующем листинге. Серверы других баз данных могут использовать иные подходы, но все они обеспечивают новое уникальное значение при вставке или обновлении.

Листинг 10.17 SQL-код для создания таблицы Authors со столбцом `timestamp`

```
CREATE TABLE [dbo].[Authors] (
    [ConcurrencyAuthorId] INT  IDENTITY (1, 1),
    [ChangeCheck]         TIMESTAMP NULL,
    [Name]                NVARCHAR (MAX) NULL
);
```

Если таблица создается EF Core, для типа столбца задается значение TIMESTAMP, если ваше свойство имеет тип byte []. Значение этого столбца будет обновляться при каждой вставке или обновлении

Мы моделируем параллельное изменение, используя код из листинга 10.18, который состоит из трех этапов:

1. Мы используем EF Core для чтения строки Authors, которую вы хотите обновить.
2. Мы используем команду SQL для обновления таблицы Authors, имитируя еще одну задачу, обновляя того же автора, которого

мы читаем. EF Core ничего не знает об этом изменении, потому что SQL обходит снимки отслеживания. (См. раздел 11.5 в главе 11 для получения подробной информации о командах SQL.)

3 В последних двух строках мы обновляем имя автора и вызываем метод `SaveChanges`, который выбрасывает исключение `DbUpdateConcurrencyException`, потому что EF Core обнаружил, что столбец ChangeCheck изменился.

Листинг 10.18 Имитация параллельного обновления сущности `ConcurrentAuthor`

```
var firstAuthor = context.Authors.First();
context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Authors SET Name = @p0"+
    " WHERE ConcurrencyAuthorId = @p1",
    firstAuthor.Name,
    firstAuthor.ConcurrencyAuthorId);
firstAuthor.Name = "Concurrency Name";
context.SaveChanges();
```

Загружает первого автора из базы данных как отслеживаемую сущность

Имитирует еще один поток/приложение, обновляющее сущность. Ничего не изменилось, кроме временной метки

Изменяет данные автора, чтобы EF Core обновил автора

Выбрасывает исключение DbUpdateConcurrencyException

Этот код аналогичен случаю, в котором мы использовали маркер параллелизма. Разница состоит в том, что временная метка обнаруживает обновление строки через уникальное значение в свойстве/столбце ChangeCheck. Можно увидеть эту разницу в следующем листинге, где приводится код SQL, который EF Core создает для обновления строки с проверкой свойства `ChangeCheck`.

Листинг 10.19 Код SQL для обновления имени автора с проверкой ChangeCheck

```
SET NOCOUNT ON;
UPDATE [Authors] SET [Name] = @p0
WHERE [ConcurrencyAuthorId] = @p1
      AND [ChangeCheck] = @p2;
SELECT [ChangeCheck]
FROM [Authors]
WHERE @@ROWCOUNT = 1
      AND [ConcurrencyAuthorId] = @p1;
```

Проверяет, что столбец ChangeCheck не изменился с тех пор, как вы прочитали сущность Author

Поскольку обновление изменит столбец ChangeCheck, EF Core необходимо прочитать его, чтобы его копия в памяти была правильной

Проверяет, была ли обновлена одна строка в последней команде. В противном случае значение ChangeCheck не будет возвращено, и EF Core узнает, что произошло параллельное изменение

Часть, касающаяся команды `UPDATE`, проверяет, имеет ли столбец ChangeCheck то же значение, что и копия, полученная при первом чтении сущности, и если это так, то выполняет обновление. Вторая часть возвращает новый столбец ChangeCheck, созданный сервером базы данных после текущего обновления, но только если была вы-

полнена команда UPDATE. Если для свойства ChangeCheck значение не возвращается, EF Core знает, что произошел конфликт параллельного доступа, и выбрасывает исключение DbUpdateConcurrencyException.

Ваш выбор между двумя подходами – маркером параллелизма и временной меткой – зависит от правил бизнес-логики. Подход с использованием маркера параллелизма обеспечивает определенную защиту свойства или свойств, в которые вы его помещаете, и срабатывает только в том случае, если свойство, помеченное как маркер параллелизма, изменяется. Подход с использованием временных меток перехватывает любое обновление этой сущности.

10.6.3 Обработка исключения DbUpdateConcurrencyException

Теперь, когда вы узнали о двух способах, используя которые, EF Core обнаруживает одновременное изменение, можно взглянуть на пример перехвата исключения DbUpdateConcurrencyException. Способ написания кода для исправления конфликта параллельного доступа зависит от причин перехвата. Пример из листинга 10.20 показывает, как перехватывать исключение DbUpdateConcurrencyException и какие данные у вас есть для принятия решений по исправлению этого исключения.

В листинге 10.20 показан метод, который вызывается после обновления сущности Book. Метод BookSaveChangesWithChecks вызывает метод SaveChanges и перехватывает исключение DbUpdateConcurrency-Exception, если оно возникает; кроме того, он использует метод HandleBookConcurrency, в который мы поместили логику для обработки исключения для сущности Book.

Листинг 10.20 Метод, который вызывается для сохранения изменений, перехватывающих конфликты параллельного доступа

```
public static string BookSaveChangesWithChecks
    (ConcurrencyDbContext context)
{
    string error = null;
    try
    {
        context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var entry = ex.Entries.Single();
        error = HandleBookConcurrency(
            context, entry);
```

Вызывается после того, как сущность Book была обновлена

Вызывает метод SaveChanges внутри try ... catch, чтобы можно было перехватить исключение DbUpdateConcurrencyException, если оно возникнет

Перехватывает исключение DbUpdateConcurrencyException и вставляет ваш код для его обработки

В этом случае вы знаете, что будет обновлена только одна сущность Book. В других случаях, возможно, потребуется обработать несколько сущностей

Вызывает метод HandleBookConcurrency, который возвращает null, если ошибка была обработана, или сообщение об ошибке, если нет

```
        if (error == null)
            context.SaveChanges();
    }
    return error;
}
```

Если конфликт обработан, необходимо вызвать метод SaveChanges, чтобы обновить Book

Возвращает сообщение об ошибке или null, если ошибки нет

Метод `BookSaveChangesWithChecks` возвращает строку, которая в случае успеха принимает значение null или сообщение об ошибке, если не может разрешить конфликт. (В этом примере мы обрабатываем конфликт обновления, но при конфликте удаления возвращаем сообщение об ошибке; см. метод `HandleBookConcurrency` из листинга 10.21.) Обратите внимание, что нужно вызвать метод `SaveChanges` повторно, только если конфликт параллелизма устранен. В противном случае метод выбросит то же самое исключение.

Метод `HandleBookConcurrency` обрабатывает конфликт обновления для сущности `Book`. В нашем распоряжении три версии данных, показанные в поле «Обработчик исключений» на рис. 10.6. В этом примере рассматривается свойство `PublishedOn`, защищенное маркером параллелизма. На рис. 10.6 показана последовательность событий и значение столбца PublishedOn на каждом этапе.

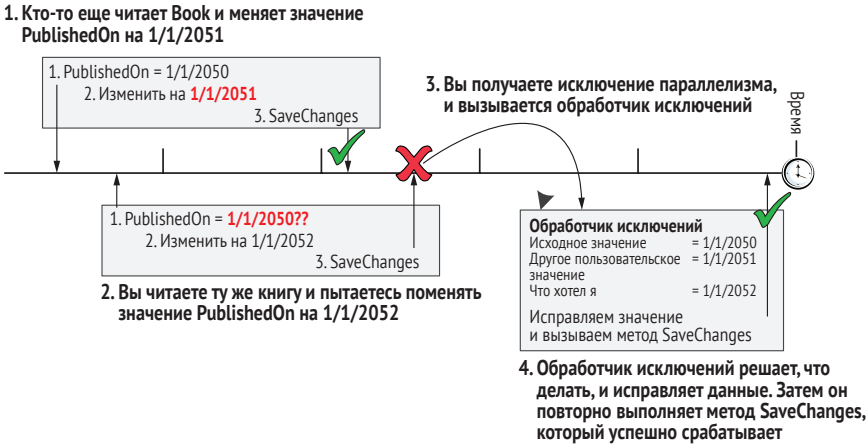
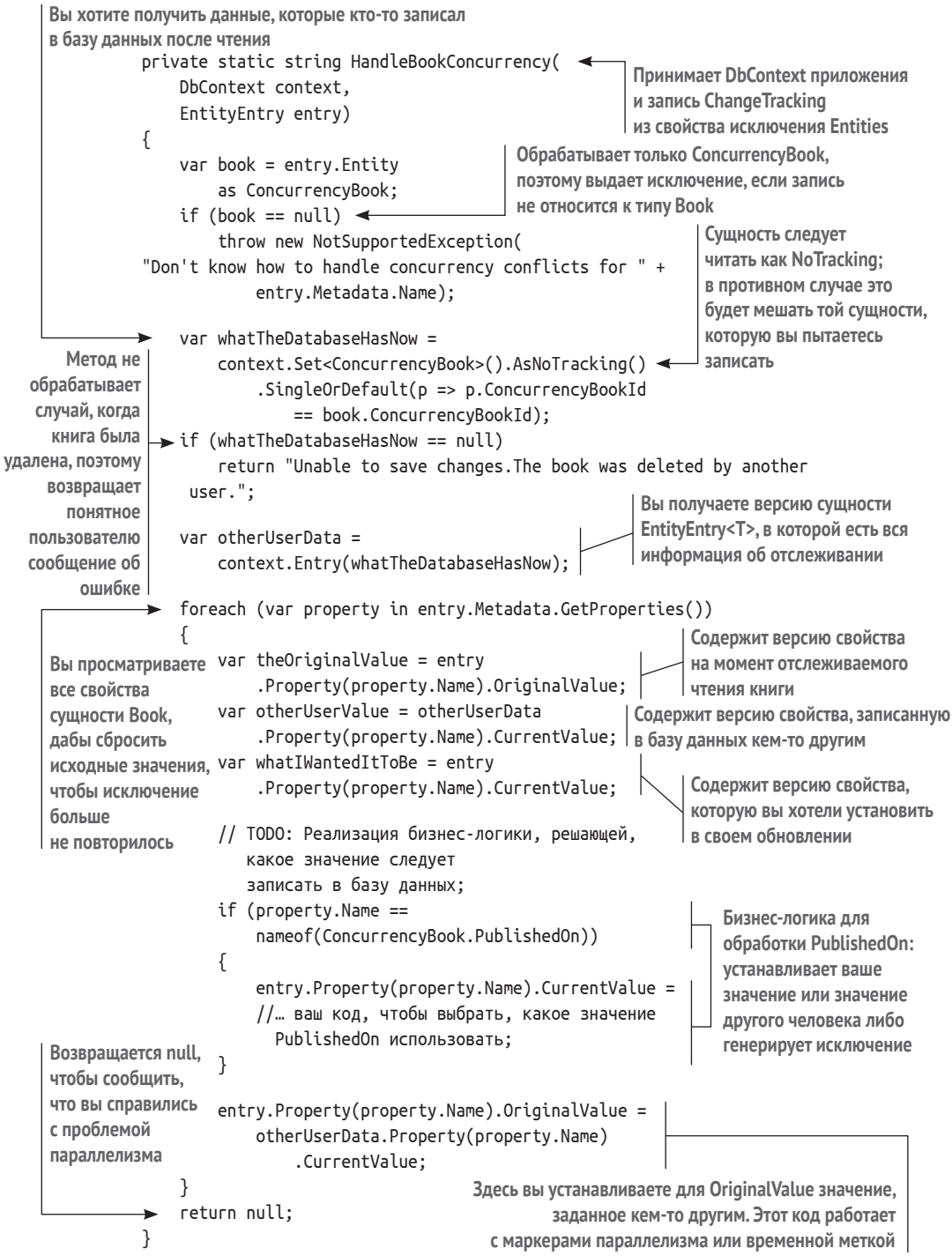


Рис. 10.6 Когда два человека обращаются к одной и той же книге, это можно обнаружить с помощью маркера параллелизма из данного примера (но этапы были бы теми же, если бы использовалась временная метка). На этапе 3 возникает исключение и вызывается обработчик исключений (см. листинг 10.21). Обработчик исключений получает копию исходной даты `PublishedOn`, которую вы получили до изменения, значение, которое другой пользователь установил для даты `PublishedOn`, и фактическое значение, которое вы хотели установить для даты `PublishedOn`

В листинге 10.21 показано содержимое обработчика исключений `HandleBookConcurrency`. В коде объявляются переменные: `originalValue`; `otherUserValue`; `whatIWantedItToBe`. Эти переменные соответствуют трем версиям данных, как показано на рис. 10.6.

Листинг 10.21 Обработка параллельного обновления для книги



Главное, что нужно изменить, – это раздел, начинающийся с комментария //TODO: Нужно поместить туда свой код для обработки параллельного обновления. Что вы туда поместите, зависит от правил бизнес-логики приложения. В разделе 10.6.4 я покажу вам проработанный пример с бизнес-логикой, но в листинге 10.21 основное внимание уделяется трем частям данных: исходные значения, значения других пользователей и нужное вам значение `PublishedOn`.

Обратите внимание, что метод `HandleBookConcurrency` также определяет случай, когда конфликт параллельного доступа вызван удалением исходной сущности `Book`. В том случае, если метод обработки параллелизма попытается заново прочитать фактическую строку в базе данных, используя первичный ключ `Book`, он не найдет ее и вернет значение `null`. Текущая реализация не обрабатывает этот случай и возвращает сообщение об ошибке, которое будет показано пользователю.

Ссылка на более сложные примеры параллелизма

Поскольку обработку параллелизма довольно сложно понять, я сделал два упрощения в описаниях в этой главе:

- метод `HandleBookConcurrency`, показанный в листинге 10.21, обрабатывает только одну сущность;
- метод `BookSaveChangesWithChecks`, показанный в листинге 10.20, предполагает, что новая проблема параллелизма не возникает, если код `HandleBookConcurrency` исправил текущую.

В реальных приложениях вам, возможно, потребуется обрабатывать несколько сущностей в обработчике параллелизма, и нельзя гарантировать, что вы не получите еще одно исключение, когда будете сохранять исправленную сущность, которая вызвала первое исключение. К счастью, в главе 15 приведены примеры решения этих проблем.

В разделе 15.5 я описываю способ хранения значений, содержащих предварительно вычисленные значения, например среднюю оценку книги, чтобы повысить производительность приложения Book App при работе с большими объемами данных. Эти дополнительные значения необходимо обновлять всякий раз, когда изменяются соответствующие сущности, но, конечно, несколько обновлений вызовут проблемы параллелизма, поэтому мне пришлось решить их.

Что касается первого упрощения (только одна сущность), взгляните на листинг 15.9, который обрабатывает несколько сущностей с проблемами параллелизма, а также различные типы такого рода проблем внутри одного класса сущности.

Что касается второго упрощения (параллелизм внутри параллелизма), см. листинг 15.8, где мы добавили цикл `do/while` вокруг вызова метода `SaveChanges`. Этот цикл означает, что код перехватит параллелизм внутри параллелизма; обработчики параллелизма учитывают данный случай.

10.6.4 Проблема с отключенным параллельным обновлением

В таких приложениях, как сайты, может возникнуть еще один сценарий параллельного обновления, который охватывает часть системы, связанную с взаимодействием с пользователем. Примеры, которые мы видели до сих пор, охватывают одновременное обновление из кода, но, если учесть человеческий фактор, проблема более вероятна и может быть более актуальной для бизнеса.

На рис. 10.7 показано, что сотруднику по имени Джон Доу повысили зарплату. Такое решение было принято начальником Джона и отделом кадров. Время между тем, как каждый из участников увидит цифру и решит, что делать, измеряется в минутах, а не в миллисекундах, но если ничего с этим не сделать, то у вас может возникнуть еще один конфликт параллельного доступа, при котором, возможно, заработная плата будет задана неверно.

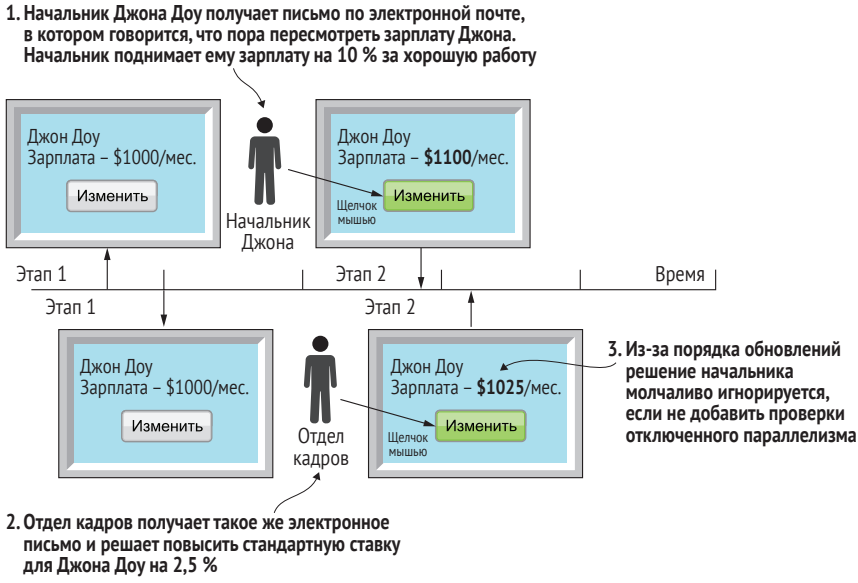


Рис. 10.7 Проблема параллелизма, возникающая в реальном времени. Ожидается пересмотр заработной платы Джона Доу, и два человека – начальник Джона и сотрудник отдела кадров – пытаются обновить сумму его зарплаты одновременно. Если не добавить проверку параллелизма, то обновление от начальника, которое пришло первым, игнорируется, что, скорее всего, не является правильным поведением

Хотя этот пример очень похож на пример конфликтов параллельного доступа из раздела 10.6.2, отличие заключается в том, как обнаруживается конфликт отключенного обновления. Чтобы обработать отключенное обновление, исходное значение защищаемого от конфликта свойства (в данном случае это `Salary`) должно быть переведено

с первого этапа отключенного обновления на второй этап. Затем второй этап должен использовать исходное свойство `Salary` при проверке конфликта параллельного доступа во время процесса обновления.

Кроме того, различается и способ обработки конфликта параллельного доступа. Обычно в случае с людьми решение о том, что должно произойти, принимает пользователь. Если возникает конфликт, пользователь видит новый экран с информацией о том, что произошло, и ему предоставляется выбор, что следует дальше сделать. Такая ситуация изменяет подход к обработке исключения `DbUpdateConcurrencyException`, в результате чего роль кода заключается скорее в диагностике, чем в устранении проблемы.

При наличии конфликта параллельного доступа пользователь видит экран с сообщением об ошибке, объясняющим, что произошло. Затем пользователю предлагается сделать выбор: принять текущее состояние или применить обновление другого пользователя, зная, что оно заменит его текущее обновление.

На рис. 10.8 показано, что происходит, когда пользователь нажимает кнопку «Изменить» после указания новой суммы зарплаты. Как видите, исходная зарплата, которую пользователь видел на первом экране, отправляется обратно с другими данными и используется при проверке параллелизма при обновлении свойства `Salary`. (См. метод `UpdateSalary` из листинга 10.24.)

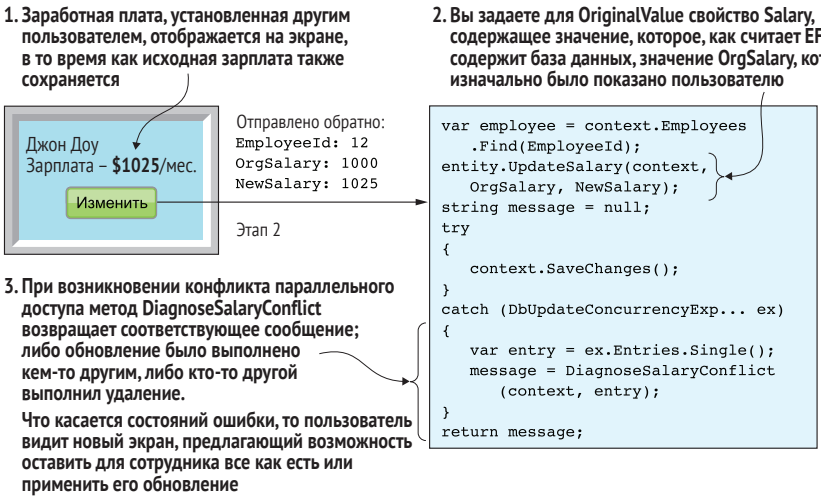


Рис. 10.8 После того как пользователь изменил зарплату и нажал кнопку «Изменить», новое и исходное значения заработной платы отправляются обратно в веб-приложение. Затем приложение вызывает метод `UpdateSalary`, показанный в листинге 10.24, который обновляет сумму зарплаты и устанавливает исходное значение, ожидаемое в базе данных при обновлении. При обнаружении конфликта параллельного доступа новый экран с соответствующим сообщением об ошибке отображается пользователю, который затем может принять существующее состояние базы данных или применить собственное обновление для сотрудника

В листинге 10.22 показан класс сущности, используемый в этом примере. Свойство `Salary` установлено как маркер параллелизма. Кроме того, мы создаем метод `UpdateSalary`, содержащий код, который необходимо выполнить для обновления свойства `Salary` таким образом, чтобы было сгенерировано исключение `DbUpdateConcurrencyException`, если значение `Salary` изменилось относительно значения, первоначально отображаемого на экране пользователя.

Листинг 10.22 Класс сущности, используемый для хранения зарплаты сотрудника с проверкой параллелизма

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    [ConcurrencyCheck]
    public int Salary { get; set; }
    public void UpdateSalary
        (DbContext context,
         int orgSalary, int newSalary)
    {
        Salary = newSalary;
        context.Entry(this).Property(p => p.Salary)
            .OriginalValue = orgSalary;
    }
}
```

Свойство Salary, заданное как маркер параллелизма атрибутом ConcurrencyCheck

Обновляет свойство Salary в отключенном состоянии

Устанавливает для свойства Salary новое значение

Устанавливает для OriginalValue, хранящего данные, считанные из базы данных, исходное значение, которое было показано пользователю на первом этапе обновления

После применения метода `UpdateSalary` к сущности `Employee` лица, чью зарплату вы хотите изменить, в блоке `try… catch` вызывается метод `SaveChanges` для обновления. Если метод `SaveChanges` выбрасывает исключение `DbUpdateConcurrencyException`, задача метода `DiagnoseSalaryConflict`, показанного в следующем листинге, заключается не в устранении конфликта, а в создании соответствующего сообщения об ошибке, чтобы пользователь мог решить, что делать.

Листинг 10.23 Возвращает различные ошибки при конфликте параллельного доступа при обновлении или удалении

```
private string DiagnoseSalaryConflict(
    ConcurrencyDbContext context,
    EntityEntry entry)
{
    var employee = entry.Entity
        as Employee;
    if (employee == null)
```

Вызывается, если возникает исключение DbUpdateConcurrencyException. Его задача не в том, чтобы решить проблему, а в том, чтобы сформировать сообщение об ошибке и предоставить варианты решения проблемы

Если сущность, которая привела к ошибке, – это не Employee, генерируется исключение, так как этот код не может обработать такой случай

```
            throw new NotSupportedException(
"Don't know how to handle concurrency conflicts for " +
                entry.Metadata.Name);

        var databaseEntity =
            context.Employees.AsNoTracking()
                .SingleOrDefault(p =>
                    p.EmployeeId == employee.EmployeeId);

        if (databaseEntity == null)
            return
$"The Employee {employee.Name} was deleted by another user. " +
$"Click Add button to add back with salary of {employee.Salary}" +
" or Cancel to leave deleted.";

        return
$"The Employee {employee.Name}'s salary was set to " +
$"{databaseEntity.Salary} by another user. " +
$"Click Update to use your new salary of {employee.Salary}" +
$" or Cancel to leave the salary at {databaseEntity.Salary}.";
    }
```

Должен читаться как NoTracking; в противном случае это будет мешать сущности, которую вы пытаетесь обновить

Вы хотите получить данные, которые кто-то записал в базу данных после вашего чтения

Проверяет наличие конфликта при удалении: сотрудник был удален, а пользователь пытается его обновить

Сообщение об ошибке для отображения пользователю с двумя вариантами действий

В противном случае эта ошибка из-за конфликта обновления, поэтому возвращаем другое сообщение об ошибке с двумя вариантами для этого случая

В листинге 10.24 показаны два метода: один для конфликта обновлений и другой для конфликта удалений. Эти методы вызываются в зависимости от типа конфликта параллельного доступа, который вы обнаружили (обновление или удаление), и только если пользователь хочет применить обновление к `Employee`.

Конфликт обновлений можно разрешить с помощью того же метода `UpdateSalary`, используемого для обычного обновления, но теперь параметр `orgSalary` – это значение зарплаты, прочитанное, когда было выброшено исключение `DbUpdateConcurrencyException`. Метод `FixDeleteSalary` используется, когда другой пользователь удалил `Employee`, а текущий пользователь хочет вернуть сущность `Employee` с новой зарплатой.

Листинг 10.24 Два метода для обработки конфликтов обновлений и удалений

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    [ConcurrencyCheck]
    public int Salary { get; set; }

    public void UpdateSalary
        (DbContext context,
         int orgSalary, int newSalary)
    {
        Salary = newSalary;
```

Устанавливаем маркер параллелизма с помощью атрибута ConcurrencyCheck

Тот же метод, который используется для обновления свойства Salary, можно использовать для исправления конфликта обновлений, но на этот раз ему будет присвоено исходное значение, которое было обнаружено при возникновении исключения DbUpdateConcurrencyException

Устанавливает новое значение для свойства Salary

```
            context.Entry(this).Property(p => p.Salary)
                .OriginalValue = orgSalary;
        }

        public static void FixDeletedSalary
        (DbContext context,
            Employee employee)
        {
            employee.EmployeeId = 0;
            context.Add(employee);
        }
    }
```

Обрабатывает конфликт удалений

Устанавливает OriginalValue. Теперь это то значение, которое содержала база данных, когда возникло исключение DbUpdateConcurrencyException

Чтобы Add сработал, ключ должен соответствовать значению среды CLR по умолчанию

Добавляет сотрудника, поскольку он был удален из базы данных и, следовательно, его нужно добавить обратно

ПРИМЕЧАНИЕ В этих примерах используется маркер параллелизма, но они также хорошо работают и с временной меткой. Чтобы использовать временную метку вместо маркера параллелизма `Salary`, используемого в этих примерах, добавьте временную метку и установите ей исходное значение перед обновлением.

Резюме

- Использование пользовательских функций SQL с EF Core для перемещения вычислений в базу данных может улучшить производительность запросов;
- настройка столбца как вычисляемого столбца SQL позволяет возвращать вычисляемое значение на основе других свойств в строке;
- EF Core предоставляет два способа установить значение по умолчанию для свойства или столбца в сущности; эти методы выходят за рамки того, чего можно достичь при установке значения по умолчанию через .NET;
- метод EF Core `HasSequence` позволяет получить предсказуемую последовательность, предоставляемую сервером базы данных, чтобы применить ее к столбцу в таблице;
- при создании или миграции базы данных за пределами EF Core необходимо настроить столбцы, которые ведут себя не так, как обычно, например сообщить EF Core, что ключ генерируется в базе данных;
- EF Core предоставляет маркеры параллелизма и временные метки для обнаружения конфликтов параллельного доступа;
- при обнаружении конфликта параллельного доступа EF Core выбрасывает исключение `DbUpdateConcurrencyException`, а затем позволяет реализовать код для обработки конфликта.

Для читателей, знакомых с EF6:

- три метода настройки значений по умолчанию, метод `HasSequence` и установка вычисляемого столбца недоступны в EF6.x;
- EF Core обрабатывает конфликт параллельного доступа так же, как и EF6.x, но Microsoft предлагает несколько незначительных изменений в способе обработки исключения `DbUpdateConcurrencyException`; см. http://mng.bz/O1VE.

11 Углубляемся в DbContext

В этой главе рассматриваются следующие темы:

- как DbContext приложения определяет изменения в отслеживаемых сущностях;
- использование метода отслеживания изменений в DbContext для ведения аудита;
- использование команд SQL через свойство DbContext `Database`;
- поиск сущностей с помощью свойства DbContext `Model`;
- обеспечение отказоустойчивости подключения к базе данных.

В этой главе рассматриваются свойства и методы, доступные в DbContext приложения. Вы уже видели некоторые из них, например методы `Add`, `Update` и `Remove`, описанные в главе 3, но здесь мы подробно рассмотрим, как они работают. Кроме того, познакомимся с другими свойствами и методами, которые не рассматривались ранее. Мы разберем методы, используемые для записи в базу данных, способы ускорить сохранение данных и способы выполнения команд SQL непосредственно в базе данных. Вы также узнаете, как получить доступ к информации о конфигурации EF Core и как ее использовать. В этой главе обсуждаются свойства DbContext для настройки свойства `State` класса сущности, в том числе что делать, если вызов метода `SaveChan-`

ges занимает слишком много времени. Но начнем мы с обзора четырех свойств из класса DbContext.

11.1 Обзор свойств класса DbContext

DbContext приложения, который наследует от класса EF Core DbContext, – ключ к доступу к базе данных. Везде, где приложению требуется EF Core, оно должно использовать экземпляр DbContext.

В этой главе основное внимание уделяется методам и открытым свойствам, которые были унаследованы от класса DbContext. Эти свойства предоставляют информацию и методы, позволяющие лучше управлять классами сущностей и их отображением в базу данных:

- ChangeTracker – обеспечивает доступ к отслеживанию изменений EF Core. Мы использовали это свойство в главе 4 для запуска проверки данных перед вызовом метода SaveChanges. Потратим немало времени на изучение свойства State класса сущности в этой главе, включая свойство ChangeTracker (раздел 11.4);
- ContextId – уникальный идентификатор экземпляра DbContext. Его основная роль – быть идентификатором корреляции для журналирования и отладки, чтобы определять, что чтение и запись были выполнены из одного и того же экземпляра DbContext;
- Database – обеспечивает доступ к трем основным группам функций:
  - контроль транзакций, описанный в разделе 4.7.2;
  - создание и миграция базы данных, описанные в главе 9;
  - команды SQL, описанные в разделе 11.5;
- Model – предоставляет доступ к модели базы данных, которую EF Core использует при подключении или создании базы данных. Раздел 11.6.2 посвящен этой теме.

11.2 Как EF Core отслеживает изменения

EF Core использует свойство State, которое присваивается всем отслеживаемым сущностям. Оно содержит информацию о том, что должно произойти с этой сущностью при вызове метода DbContext SaveChanges.

ОПРЕДЕЛЕНИЕ Как вы, возможно, помните из главы 2, *отслеживаемые сущности* – это экземпляры сущностей, полученные из базы данных с помощью запроса, который не включал в себя метод AsNoTracking. Также после использования экземпляра сущности в качестве параметра для методов EF Core, таких как Add, Update, Remove или метода Attach, он становится отслеживаемым.

Свойство `State`, принимающее значение перечисления `EntityState`, обычно устанавливается функцией отслеживания изменений внутри EF Core, и в этом разделе мы изучим все способы изменения значения этого свойства. В главе 3 было дано краткое описание свойства `State`, но было пропущено множество его возможностей, особенно тех, что связаны со связями, а также дополнительные методы, которые рассматриваются в этом разделе. В следующем списке, взятом из главы 3, перечислены возможные значения свойства `State`, доступ к которому осуществляется с помощью команды EF Core `context.Entry(myEntity).State`:

- `Added` – сущности пока еще нет в базе данных. Метод `SaveChanges` вставляет ее;
- `Unchanged` – сущность находится в базе данных и не была изменена. Метод `SaveChanges` игнорирует ее;
- `Modified` – сущность находится в базе данных и была изменена. Метод `SaveChanges` обновляет ее;
- `Deleted` – сущность находится в базе данных, но должна быть удалена. Метод `SaveChanges` удаляет ее;
- `Detached` – сущность не отслеживается. Метод `SaveChanges` ее не видит.

На рис. 11.1 показано изменение свойства экземпляра сущности `State` без каких-либо связей от создания и до удаления из базы данных. Это наглядный пример для изучения значений, которые могут быть у этого свойства.

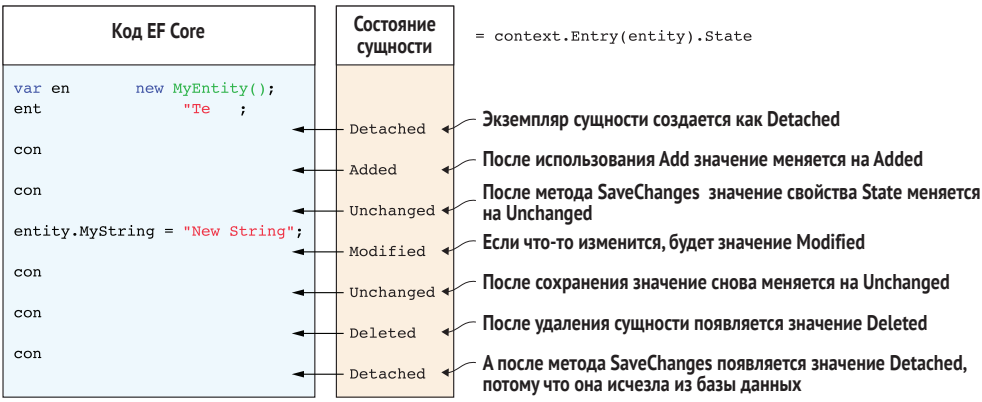


Рис. 11.1 В коде слева используются все стандартные способы создания, обновления и удаления данных в базе данных. В правом столбце показано состояние сущности на каждом из этапов

Когда у вас есть сущность в состоянии `Modified`, еще один логический флаг, `IsModified`, вступает в игру. Он определяет, какое из свойств, скалярных и навигационных, изменилось в сущности. Доступ к свойству `IsModified` для скалярного свойства можно получить с помощью

```
context.Entry(entity).Property("PropertyName").IsModified,
```

а для навигационных свойств – через

```
context.Entry(entity).Navigation("PropertyName").IsModified
```

Эти два способа доступа к свойству `IsModified` предоставляют флаг для каждого свойства, теневого свойства или резервного поля, чтобы определить, что изменилось, если для свойства сущности `State` установлено значение `Modified`.

11.3 Обзор команд, которые изменяют свойство сущности State

На рис. 11.1 показана простая сущность, но когда используются связи, настройки свойства `State` усложняются. В следующих подразделах представлены команды, которые могут изменить это свойство и связи сущности.

Подход EF Core к установке этого свойства был пересмотрен на основе отзывов из предыдущих версий EF (EF6.x и EF Core 1.x), чтобы установить свойство `State` связанных сущностей в наиболее «естественное» состояние, основываясь на определенных критериях, особенно когда вы добавляете или обновляете сущность со связями. Например, если вы используете метод `Add` для добавления новой сущности со связями в базу данных, EF Core решит, следует ли установить для свойства `State` значение `Added` или `Modified`, в зависимости от того, отслеживает ли ее EF Core. В целом это работает корректно для большинства вызовов метода `Add`, но знание того, как EF Core решает, как установить значение свойства `State`, поможет вам для работы в более сложных сценариях.

> **EF6** Настройка свойства сущности `State` в EF Core отличается от того, как это происходит в EF6.x, когда вы используете такие методы, как `Add` и `Remove`. В этой главе описывается, как EF Core устанавливает значение свойства `State`. Если вас интересуют изменения, связанные с EF6.x, рекомендую посетить страницу http://mng.bz/YA8A.

Для начала посмотрите на табл. 11.1, где перечислены команды и действия, изменяющие свойство `State`.

> **ПРИМЕЧАНИЕ** Методы `SaveChanges` и `SaveChangesAsync` сбрасывают значение свойства `State` всех отслеживаемых классов сущностей на `Unchanged`. Эта тема рассматривается в разделе 11.4.

Таблица 11.1 Все команды и действия EF Core, которые могут изменить свойство отслеживаемой сущности State. Здесь показан пример каждой команды и действия и конечное свойство State

Команда/действие	Пример	Конечное свойство State
Add/AddRange	context.Add(entity);	Added
Remove/RemoveRange	context.Remove(entity);	Deleted
Изменение свойства	entity.MyString = "hello";	Modified
Update/UpdateRange	context.Update(entity);	Modified
Attach/AttachRange	context.Attach(entity);	Unchanged
Установка State напрямую	context.Entry(entity).State = …	Переданное значение State
Установка State через TrackGraph	context.ChangeTracker.TrackGraph(…	Переданное значение State

В таблице показано, что происходит с отдельным классом сущности без связей, но большинство команд также используют рекурсивный поиск навигационных свойств, чтобы найти все затронутые классы сущностей. Любая команда, выполняющая рекурсивный поиск, будет отслеживать каждый используемый класс сущности и устанавливать значение для свойства State.

Мы уже сталкивались с большинством этих команд и действий, но некоторые команды, например Attach и TrackGraph, пока еще не рассматривались. В этом разделе мы изучим каждую команду и действие. Если команда или действие уже были описаны, то раздел будет коротким. Новые команды и действия описываются подробнее.

11.3.1 Команда Add: вставка новой строки в базу данных

Методы Add и AddRange используются для создания новой сущности в базе данных путем установки для свойства State значения Added. В разделе 3.2 описывается метод Add, а в разделе 6.2.2 приводится подробное пошаговое описание добавления класса сущности со связями. Обобщим:

- для свойства сущности State задается значение Added;
- метод Add просматривает все сущности, связанные с добавленной сущностью:
  - если связь в настоящее время не отслеживается, она начинает отслеживаться, и для свойства связанной сущности State задается значение Added;
  - если связь отслеживается, используется ее текущее свойство State, только если не было требования изменить или установить внешний ключ, в этом случае для свойства State задается значение Modified.

Кроме того, методы AddAsync и AddRangeAsync доступны для сущностей, использующих генератор значений (см. раздел 10.3.3), чтобы установить значение по умолчанию для свойства. Если у генератора значений есть метод NextAsync, вы должны использовать методы AddAsync и AddRangeAsync при добавлении этой сущности.

11.3.2 Метод Remove: удаление строки из базы данных

Методы `Remove` и `RemoveRange` удаляют сущность из базы данных путем записи в свойство `State` значения `Deleted`. Метод `Remove` описан в разделе 3.5, а в разделе 8.8.1 описаны способы удаления, поддерживаемые EF Core. В этом разделе мы рассмотрим только то, что происходит со свойством `State` сущности, которую вы удаляете, и свойством `State` всех ее связей. Если у удаленной сущности есть связи, которые загружаются и отслеживаются, то значение свойства `State` для каждой связи будет одним из следующих:

- `State == Deleted` – для обязательной зависимой связи, например класса сущности `Review`, связанного с классом сущности `Book`;
- `State == Modified` – для необязательной зависимой связи, в которой внешний ключ допускает значение NULL. В этом случае необязательная связь не удаляется, но внешний ключ, связанный с удаленной сущностью, получает значение `null`;
- `State == Unchanged` – результат удаления зависимой сущности, которая связана с главным классом. При удалении зависимой сущности в основных и внешних ключах класса ничего не меняется.

ПРИМЕЧАНИЕ Если прочитать класс сущности, добавить обязательную зависимую связь, а затем удалить класс, то можно получить несогласованные состояния свойств `State`. На короткое время у обязательной зависимой связи это свойство будет иметь значение `Added`, потому что это наиболее логичное значение на тот момент.

Но, кроме свойства `State` связей, загруженных с удаленным классом сущности, другой параметр имеет приоритет: поведение `OnDelete` удаленного класса сущности. Если для поведения `OnDelete` установлено значение `Cascade`, которое является значением по умолчанию для обязательной зависимой связи, все обязательные зависимые связи удаленного класса сущности будут удалены. Пожалуйста, обратитесь к разделу 8.8.1, где приводится более подробное объяснение.

11.3.3 Изменение класса сущности путем изменения данных в нем

Есть одна умная вещь, которую может сделать EF Core, – автоматически определить, что вы изменили данные в классе сущности, и превратить это изменение в обновление базы данных. Эта особенность делает обновления простыми с точки зрения разработчика, но требует работы со стороны EF Core. Правила таковы:

- чтобы EF Core обнаружил изменение, сущность должна отслеживаться. Сущности отслеживаются, если вы читаете их без метода `AsNoTracking` в запросе или когда вызываете метод `Add`, `Update`, `Remove` либо `Attach` с классом сущности в качестве параметра;

- при вызове методов `SaveChanges` и `SaveChangesAsync` по умолчанию EF Core выполняет метод `ChangeTracker.DetectChanges`, который сравнивает данные текущей сущности со снимком отслеживания сущности. Если какие-либо свойства, резервные поля или теневые свойства отличаются, то для свойства `State` устанавливается значение `Modified`, а для свойств, резервных полей или теневых свойств устанавливается значение `IsModified`.

На рис. 11.2 показано, как EF Core может обнаружить изменение. В этом примере единственное изменение касается одного из свойств в первой сущности `Book`.

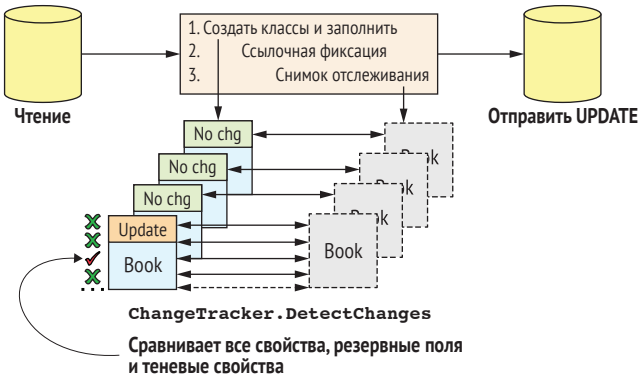


Рис. 11.2 Метод `SaveChanges` вызывает метод `ChangeTracker.DetectChanges`, который сравнивает каждую отслеживаемую сущность с соответствующим снимком отслеживания для обнаружения любых различий между ними. Метод `ChangeTracker.DetectChanges` сравнивает все данные, которые отображаются в базу данных. В этом примере было изменено только одно свойство в первой сущности `Book`. На рисунке она отмечена галочкой, а сверху стоит заголовок «Обновить»

11.3.4 Изменение класса сущности путем вызова метода Update

В разделе 11.3.3 показано, что EF Core может обнаруживать изменения в классе сущности автоматически. Однако в главе 3 мы столкнулись с внешней системой, которая вернула класс сущности в формате JSON для обновления (см. рис. 11.3, копию рис. 3.3), но этот класс сущности не отслеживался. В этом случае метод `ChangeTracker.DetectChanges` не сработает, потому что у EF Core нет снимка отслеживания, чтобы провести сравнение. В подобных случаях можно использовать методы `Update` и `UpdateRange`.

Метод `Update` дает EF Core команду обновить все свойства и столбцы в этой сущности путем установки для данного свойства сущности `State` значения `Modified`, а для свойства `IsModified` – значения `true` у всех нереляционных свойств, включая любые внешние ключи. В результате у строки в базе данных будут обновлены все столбцы.

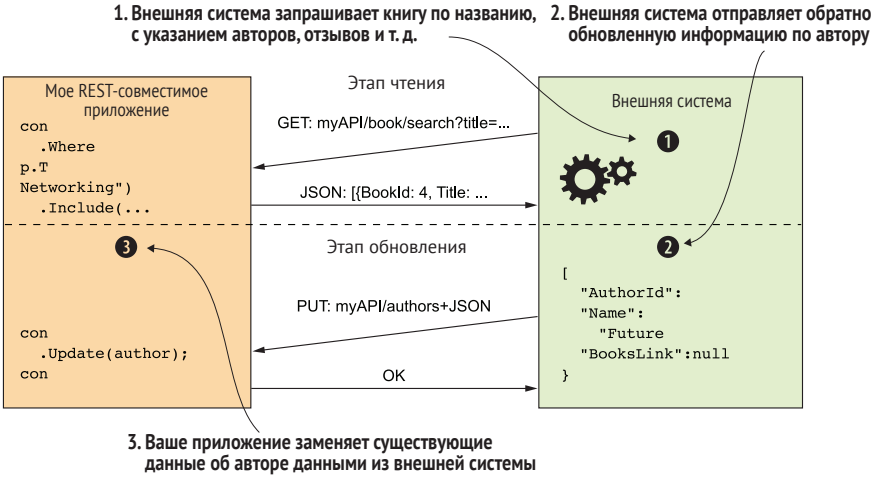


Рис. 11.3 Пример отключенного обновления, в котором мы заменяем всю информацию из базы данных новыми данными. Внешняя система справа возвращает содержимое класса `Author` в формате JSON. Приложение ASP.NET Core слева преобразует отправленный JSON обратно в класс сущности `Author`, а принимающий код использует команду EF Core `Update` для обновления таблицы Authors в базе. Эта команда обновляет все свойства, резервные поля и теневые свойства в воссозданном классе сущности

Если у типа сущности, используемого в методе `Update`, есть загруженные связи, то метод `Update` будет рекурсивно просматривать каждый связанный класс сущности и устанавливать значения для свойства `State`. Правила настройки этого свойства для связанного класса сущности зависят от того, генерируется ли первичный ключ сущности со связями базы данных и задан ли он (его значение не является значением по умолчанию для типа .NET). Тогда:

- если ключ, который *генерируется* базой данных, отличен от значения по умолчанию, EF Core допускает, что сущность со связью уже находится в базе данных, и установит для свойства `State` значение `Modified`, если необходимо установить внешний ключ; в противном случае будет значение `Unchanged`;
- если ключ не *генерируется* базой данных или является значением по умолчанию, EF Core предполагает, что это новая сущность со связью, и установит для ее свойства `State` значение `Added`.

Все это звучит довольно сложно, но EF Core обычно устанавливает для свойства `State` наиболее подходящее значение.

Например, если вы добавите существующую запись в связи класса сущности, то для свойства `State` будет установлено значение `Updated`, но если вы добавите новую запись, то для свойства `State` будет установлено значение `Added`.

11.3.5 Метод Attach: начать отслеживание существующего неотслеживаемого класса сущности

Методы `Attach` и `AttachRange` полезны, если у вас уже есть класс сущности с актуальными данными и вы хотите, чтобы они отслеживались. После того как вы прикрепляете (Attach) сущность, она становится отслеживаемой, и EF Core предполагает, что ее содержимое соответствует текущему состоянию базы данных. Это поведение хорошо подходит для воссоздания сущностей со связями, которые были сериализованы, а затем десериализованы в сущность, но только если сущности записываются обратно в ту же базу данных, поскольку первичный и внешний ключи должны совпадать.

> **ПРЕДУПРЕЖДЕНИЕ** Сериализация и последующая десериализация экземпляра класса сущности, который использует теневые свойства, требуют особой обработки с помощью метода `Attach`. Теневые свойства не являются частью класса, поэтому они будут потеряны при сериализации. Необходимо сохранять и восстанавливать любые теневые свойства, особенно внешние ключи после вызова этого метода.

Когда вы прикрепляете сущность, она становится отслеживаемой сущностью, но без затрат на ее загрузку из базы данных. Метод `Attach` делает это, устанавливая для свойства `State` значение `Unchanged`. Как и в случае с методом `Update`, установка свойства `State` для связей зависит от того, генерируется ли первичный ключ сущности базой данных и задан ли он:

- *ключ генерируется базой данных, и у него установлено значение по умолчанию* – EF Core предполагает, что это новая сущность, и установит для свойства `State` значение `Unchanged`;
- *ключ не генерируется базой данных, или его значение отличается от значения по умолчанию* – EF Core предполагает, что сущность со связями уже находится в базе данных, и установит для свойства `State` значение `Added`.

Если вы не уверены, использовать ли в своем коде методы `Attach` или `Update`, рекомендую прочитать статью Артура Виккерса «Обязательно вызывайте Update, когда это необходимо!» (http://mng.bz/G68O).

11.3.6 Установка свойства сущности State напрямую

Еще один способ задать значение свойства `State` – сделать это вручную, установив нужное вам состояние. Такой способ полезен, когда у сущности много связей и нужно конкретно решить, какое состояние для каждой связи вам нужно. В разделе 11.3.7 показан хороший пример.

Поскольку свойство `State` сущности – это свойство с доступом на чтение/запись, его можно изменить. В следующем фрагменте кода для свойства `State` экземпляра `myEntity` установлено значение `Added`:

```
context.Entry(myEntity).State = EntityState.Added;
```

Кроме того, можно установить значение флага `IsModified` для свойства в сущности. В следующем фрагменте кода для флага `IsModified` свойства `MyString` задано значение `true`, в результате чего значение свойства `State` становится `Modified`:

```
var entity = new MyEntity();
context.Entry(entity).Property("MyString").IsModified = true;
```

ПРИМЕЧАНИЕ Если сущность не отслеживалась до того, как вы установили значение свойства `State`, то после она становится отслеживаемой.

11.3.7 TrackGraph: обработка отключенных обновлений со связями

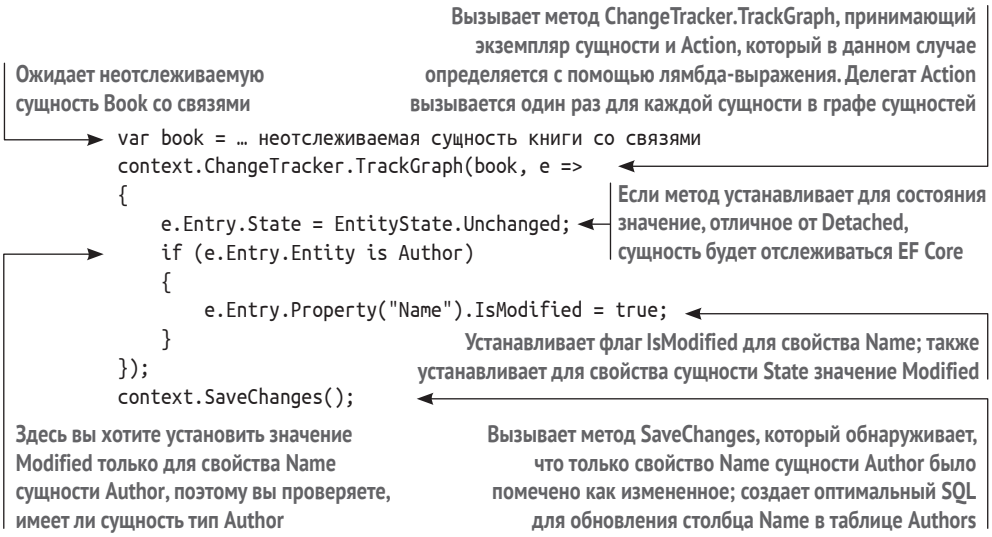
Метод `TrackGraph` полезен, если у вас есть неотслеживаемая сущность со связями и вам нужно установить правильное состояние для каждой сущности. Он проходит по всем ссылкам в сущности, вызывая делегат, указанный вами для каждой сущности, которую он находит. Этот метод полезен, если у вас есть группа связанных сущностей при отключенной ситуации (скажем, с помощью некой формы сериализации) и вы хотите изменить только часть загруженных данных.

EF6 Метод `TrackGraph` – долгожданное дополнение к EF Core. У него нет аналога в EF6.x.

Рассмотрим простой пример с REST API из главы 3, где было обновлено свойство автора `Name`. Тогда внешняя система отправляла только данные сущности `Author`. В этом примере внешняя система отправит обратно всю книгу со всеми ее связями, но ей по-прежнему нужно, чтобы вы обновили только свойство `Name` в каждом связанном классе сущности `Author`.

В листинге 11.1 показан код, который может потребоваться для обхода экземпляра сущности `Book`, который мы воссоздали из JSON (неотслеживаемой сущности). Метод `TrackGraph` вызовет лямбда-выражение `Action`, указанное в качестве второго параметра, для каждой сущности, начиная с экземпляра сущности `Book`; после этого оно будет вызвано для всех доступных ему экземпляров навигационных свойств.

Листинг 11.1 Использование метода TrackGraph для установки свойств State и IsModified у каждой сущности



Метод `TrackGraph` просматривает сущность, указанную в качестве первого параметра, и все сущности, получаемые при обходе навигационных свойств. Обход является рекурсивным, поэтому навигационные свойства любых обнаруженных сущностей также будут сканироваться. Делегат `Action`, который вы передаете в качестве второго параметра, вызывается для каждой обнаруженной неотслеживаемой сущности (`State == Detached`) и может устанавливать значение свойства `State` для отслеживания EF Core. Если значение свойства `State` не было задано, то сущность остается в состоянии `Detached` (т. е. она не отслеживается EF Core). Кроме того, метод `TrackGraph` будет игнорировать все посещаемые им сущности, которые отслеживаются в настоящее время.

Хотя вы по-прежнему можете использовать для этой цели команду `Update`, это будет неэффективно, поскольку она обновит все таблицы и столбцы в связях книги, а не только имена авторов. Метод `ChangeTracker.TrackGraph` обеспечивает более эффективный подход.

На рис. 11.4 показаны пример «изменить только имя автора» и внешняя система, возвращающая сериализованную версию сущности `Book`. Использование метода `TrackGraph` позволяет выбрать конкретную сущность и свойство, для которых нужно установить новое значение свойства `State`; в данном случае вы устанавливаете значение `IsModified` для свойства `Name` в любом классе сущности `Author` среди навигационных свойств сущности `Book`.

В результате выполнения этого кода только для свойства `State` экземпляра сущности `Author` будет установлено значение `Modified`, тогда как у всех остальных типов сущностей это свойство сохранит значение `Unchanged`. Кроме того, значение флага `IsModified` устанавливается только в свойстве `Name` класса сущности `Author`. В этом при-

мере показана разница между использованием метода `Update` и кода `TrackGraph`, уменьшающего количество обновлений базы данных: метод `Update` обновит 20 столбцов (19 из них без необходимости), тогда как код `TrackGraph` изменит только один столбец.

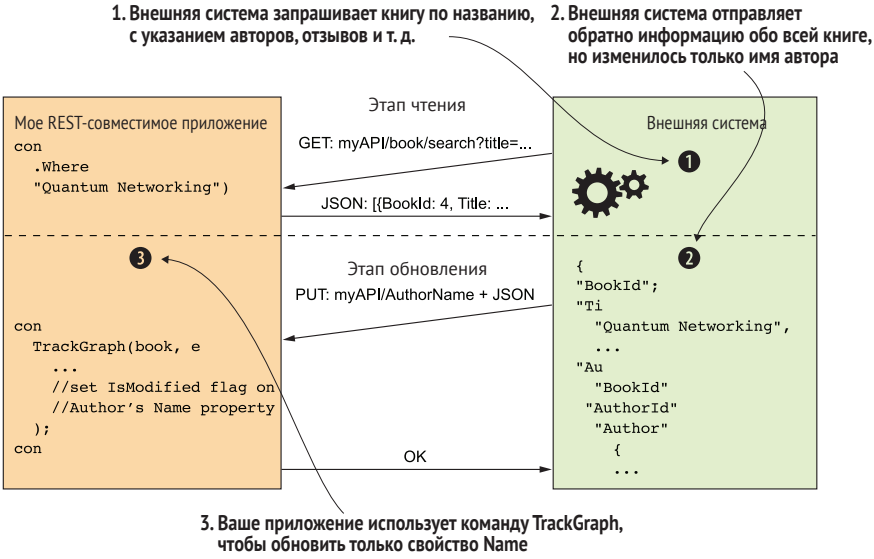


Рис. 11.4 Внешняя система, которая запрашивает конкретную книгу и получает JSON, содержащий книгу и все ее связи. Для обновления имен авторов внешняя система отправляет обратно весь исходный JSON с измененными именами, но сообщает вашему приложению, что ей необходимо изменить только имя автора. Приложение использует метод `ChangeTracker.TrackGraph`, чтобы установить для всех классов значение `State` в `Unchanged`, но устанавливает флаг `IsModified` свойства `Name` в классе сущности `Author`

11.4 Метод SaveChanges и как он использует метод ChangeTracker.DetectChanges

Раздел 11.3 был посвящен установке свойства `State` отслеживаемых сущностей, чтобы при вызове метода `SaveChanges` (или `SaveChangesAsync`) к базе данных применялись правильные обновления. В этом разделе мы рассмотрим:

- как метод `SaveChanges` определяет обновления с помощью метода `ChangeTracker.DetectChanges`;
- что делать, если выполнение метода `ChangeTracker.DetectChanges` занимает слишком много времени;
- как использовать свойство `State` отслеживаемой сущности для регистрации любых изменений;
- как подписаться на события `StateChanged` в EF Core.

11.4.1 Как метод SaveChanges находит все изменения состояния

В то время как состояния `Added` и `Deleted` настраиваются командами EF Core, подход «изменить свойство» (раздел 11.3.3) при работе с обновлениями основан на сравнении каждого класса сущности с его снимком отслеживания. Для этого метод `SaveChanges` вызывает метод `DetectChanges`, доступ к которому осуществляется через свойство `ChangeTracker`.

На рис. 11.5 (из раздела 11.3.3) показан пример, в котором были прочитаны четыре сущности `Book` и одно свойство, `PublishedOn`, было изменено в первом экземпляре сущности.

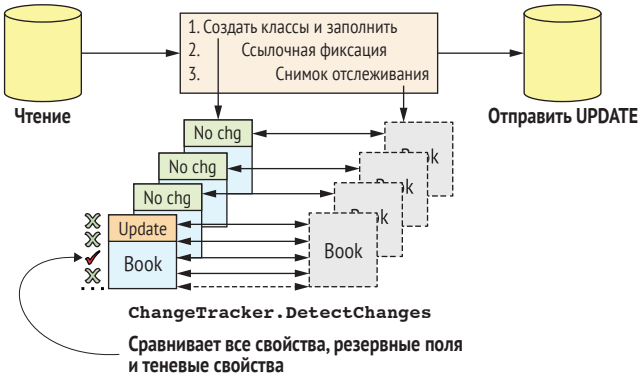


Рис. 11.5 Метод `SaveChanges` вызывает метод `ChangeTracker.DetectChanges`, который сравнивает каждую отслеживаемую сущность со снимком отслеживания, чтобы обнаружить любые различия между ними. Метод `ChangeTracker.DetectChanges` сравнивает все данные, отображенные в базу данных

Этот процесс упрощает обновление для вас как для разработчика; вы обновляете только свойство, резервное поле или теневое свойство, и изменение будет обнаружено. Но если у вас много сущностей с большим количеством данных, то процесс может замедлиться. В разделе 11.4.2 показано решение, которое можно использовать, когда выполнение метода `ChangeTracker.DetectChanges` занимает слишком много времени.

11.4.2 Что делать, если метод ChangeTracker.DetectChanges занимает слишком много времени

В некоторых приложениях может загружаться большое количество отслеживаемых сущностей. Например, при выполнении математического моделирования или создании приложения с искусственным интеллектом хранение большого количества данных в памяти может быть единственным способом достичь нужного уровня производительности.

Проблема состоит в том, что у вас большое количество отслеживаемых экземпляров сущностей и/или ваши сущности содержат много данных. В этом случае вызов метода `SaveChanges` или `SaveChangesAsync` может замедлиться. Если вы сохраняете много данных, то, скорее всего, такая медлительность вызвана обращениями к базе данных. Но если вы сохраняете только небольшой объем данных, то любое падение производительности, вероятно, связано со временем, которое требуется методу `ChangeTracker.DetectChanges` для сравнения каждого экземпляра класса сущности с соответствующим снимком отслеживания.

EF Core предлагает несколько способов заменить метод `ChangeTracker.DetectChanges` альтернативным способом обнаружения изменений. Эти функции обнаруживают отдельные обновления данных в классах сущностей, исключая любые сравнения данных, которые не были изменены. Например, тест по сохранению 100 000 крошечных сущностей без изменений занял 350 мс с обычным методом `ChangeTracker.DetectChanges`, тогда как подход, где изменения обнаруживаются с помощью вспомогательного класса, занял 2 мс для тех же данных.

Таблица 11.2 Сравнение четырех подходов, которые можно использовать, чтобы метод `ChangeTracker.DetectChanges` не проверял сущность, тем самым экономя время

Что	Плюсы	Минусы
`INotifyPropertyChanged`	▪ Может изменять только медленные сущности. ▪ Обрабатывает исключения параллелизма	▪ Необходимо редактировать каждое свойство
`INotifyPropertyChanged` и `INotifyPropertyChanging`	▪ Может изменять только медленные сущности. ▪ Нет снимка отслеживания, поэтому использует меньше памяти	▪ Необходимо редактировать каждое свойство
Отслеживание изменений через прокси (функция EF Core 5) `INotifyPropertyChanged`	▪ Легко писать код; к каждому свойству нужно добавить ключевое слово `virtual`. ▪ Обрабатывает исключения параллелизма	▪ Необходимо изменить *все* типы сущностей, чтобы использовать прокси
Отслеживание изменений через прокси (функция EF Core 5) `INotifyPropertyChanged` и `INotifyPropertyChanging`	▪ Легко писать код; нужно добавить ключевое слово `virtual`. ▪ Нет снимка отслеживания, поэтому использует меньше памяти	▪ Необходимо изменить *все* типы сущностей, чтобы использовать прокси. ▪ Придется создать новый класс сущности, используя метод `CreateProxy<T>`

В целом отслеживание изменений через прокси проще реализовать, однако нужно внести изменения во все свои классы сущностей, чтобы использовать эту функцию. Но если вы обнаружите проблему с производительностью метода `SaveChanges` в существующем приложении, то изменение всех классов сущностей может стать слишком обременительным. По этой причине я сосредоточусь на первом подходе, `INotifyPropertyChanged`, который легко добавить к нескольким

классам сущностей, имеющим проблемы, и последнем подходе, отслеживание изменений через прокси, который проще, но требует, чтобы вы использовали его во всем приложении.

ПЕРВЫЙ ПОДХОД: INotifyPropertyChanged

EF Core поддерживает интерфейс `INotifyPropertyChanged` в классе сущности, чтобы обнаружить, изменилось ли какое-либо свойство. Он уведомляет EF Core о том, что свойство изменилось, но вы должны вызвать событие `PropertyChanged`, а это означает, что метод `ChangeTracker.DetectChanges` не используется.

Чтобы использовать интерфейс `INotifyPropertyChanged`, необходимо создать вспомогательный класс `NotificationEntity`, показанный в следующем листинге. Этот класс предоставляет метод `SetWithNotify`, который вызывается при изменении любого свойства в классе сущности.

Листинг 11.2 Вспомогательный класс `NotificationEntity`, от которого наследует класс сущности NotifyEntity

```
public class NotificationEntity : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void SetWithNotify<T>(T value, ref T field,
        [CallerMemberName] string propertyName = "")
    {
        if (!Object.Equals(field, value))
        {
            field = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Автоматически получает свойство propertyName, используя System.Runtime.CompilerServices

Устанавливает новое значение для поля

Только если field и value отличаются, вы устанавливаете значение поля и вызываете событие

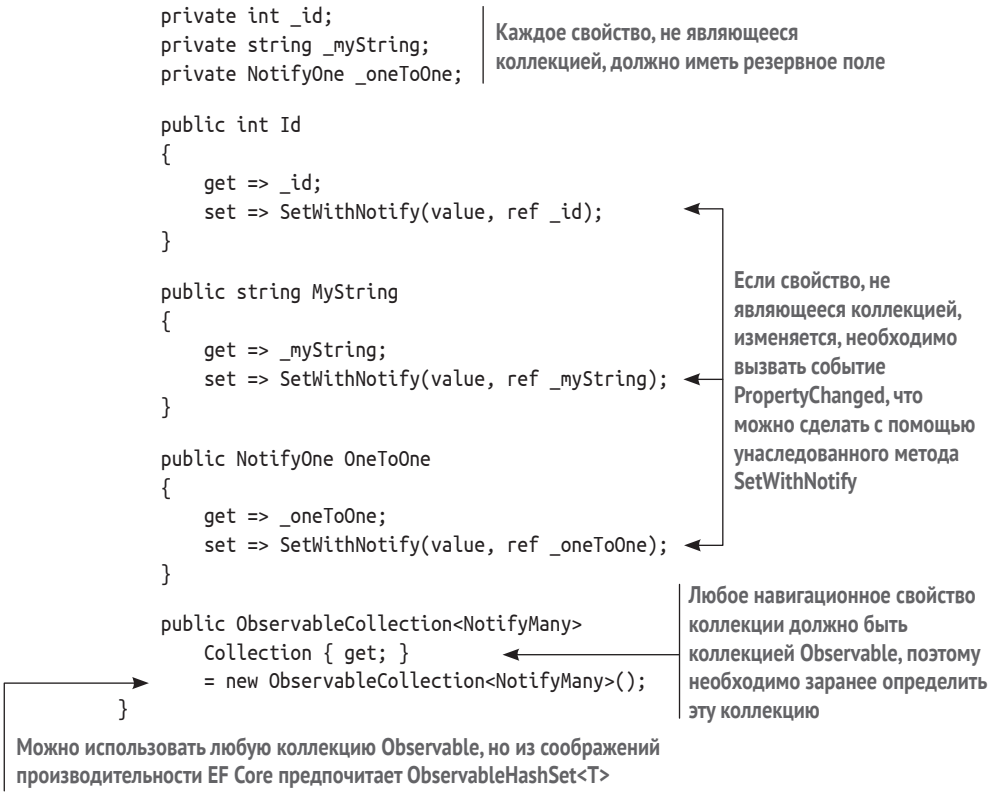
... именем свойства

Вызывает событие PropertyChanged, но использует оператор ?., чтобы предотвратить ошибку, когда создается новая сущность, а PropertyChangedEventHandler не был инициализирован EF Core...

В следующем листинге показан класс сущности `NotifyEntity`, который наследует от класса `NotificationEntity`, показанного в листинге 11.2. Вы должны вызывать метод `SetWithNotify` всякий раз, когда изменяется свойство, не являющееся коллекцией. В случае с коллекциями необходимо использовать `ObservableCollection`, чтобы вызвать событие, когда навигационное свойство коллекции изменяется.

Листинг 11.3 Класс `NotifyEntity`, использующий класс `NotificationEntity` для событий

```
public class NotifyEntity : NotificationEntity
{
```



После того как вы определили свой класс сущности для использования интерфейса `INotifyPropertyChanged`, нужно установить стратегию отслеживания для этого класса в `ChangedNotifications` (листинг 11.4). Эта конфигурация велит EF Core не обнаруживать изменения, используя метод `ChangeTracker.DetectChanges`, потому что уведомления о любых изменениях будут поступать через события `INotifyPropertyChanged`. Чтобы настроить события `INotifyPropertyChanged` для одного класса сущности, используется команда Fluent API.

Листинг 11.4 Установка стратегии отслеживания для одной сущности в `ChangedNotifications`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
      .Entity<NotifyEntity>()
      .HasChangeTrackingStrategy(
          ChangeTrackingStrategy.ChangedNotifications);
}
```

ВТОРОЙ И ТРЕТИЙ ПОДХОДЫ

Я не рассматриваю второй подход (-change и -changing события), но отличия от первого подхода следующие:

- класс `NotificationEntity` должен содержать -change и -changing события;
- используется другой параметр `ChangeTrackingStrategy`, например `ChangingAndChangedNotifications`.

Также не рассматривается третий подход (отслеживание изменений через прокси, `INotifyPropertyChanged`), который работает аналогично тому, как прокси с отложенной загрузкой работает с виртуальными свойствами.

Вместо этого я рассмотрю последний подход (описанный далее), который обрабатывает `INotifyPropertyChanged` и `INotifyPropertyChanging`. Главное отличие состоит в том, что в третьем подходе можно создать экземпляр класса сущности с помощью обычного конструктора, тогда как последний подход требует использования метода `CreateProxy<TEntity>` для создания класса сущности.

ПОСЛЕДНИЙ ПОДХОД: ОТСЛЕЖИВАНИЕ ИЗМЕНЕНИЙ ЧЕРЕЗ ПРОКСИ

В последнем подходе используется отслеживание изменений через прокси с помощью событий `INotifyPropertyChanged` и `INotifyPropertyChanging`, появившихся в EF Core 5. Эти события добавляются к подходу, где используются прокси с отложенной загрузкой и виртуальные свойства, как описано в разделе 2.4.4. Чтобы использовать этот подход, нужно сделать следующее:

- измените все классы сущностей, чтобы у них были виртуальные свойства;
- используйте тип коллекции `Observable` для навигационных свойств коллекции;
- измените код, создающий новый экземпляр класса сущности, чтобы использовать метод `CreateProxy<TEntity>`;
- добавьте библиотеку NuGet `Microsoft.EntityFrameworkCore.Proxies`;
- добавьте метод `UseChangeTrackingProxies` при создании настроек DbContext приложения.

Начнем с рассмотрения структуры класса сущности, необходимого для использования отслеживания изменений через прокси, как показано в следующем листинге.

Листинг 11.5 Пример класса сущности, настроенного на использование отслеживания изменений через прокси

```
public class ProxyMyEntity
{
    public virtual int Id { get; set; }
    public virtual string MyString { get; set; }
    public virtual ProxyOptional ProxyOptional { get; set; }

    public virtual ObservableCollection<ProxyMany>
```

Все свойства должны быть виртуальными

```
        Many { get; set; }
        = new ObservableCollection<ProxyMany>();
}
```

Для навигационных свойств коллекции необходимо использовать тип коллекции Observable

Если вы читаете класс сущности, используя запрос, то отслеживание изменений через прокси добавит дополнительный код для создания событий `INotifyPropertyChanged` и `INotifyPropertyChanging` при изменении свойства. Но если вы хотите создать новый класс сущности, то не можете использовать обычную команду, например `new Book()`. Вместо этого нужно использовать метод `CreateProxy<TEntity>`. Например, если вы хотите добавить новую версию класса `ProxyMyEntity`, показанного в листинге 11.5, то следует написать:

```
var entity = context.CreateProxy<ProxyMyEntity>();
entity.MyString = "hello";
context.Add(entity);
context.SaveChanges();
```

Нужно использовать метод `CreateProxy<TEntity>` (первая строка предыдущего фрагмента кода); в противном случае EF Core не сможет обнаружить событие изменения. (Не волнуйтесь. Если вы забудете это сделать, то EF Core сгенерирует исключение с полезным сообщением.)

Последняя часть – убедиться, что пакет NuGet `Microsoft.EntityFrameworkCore.Proxies` загружен, а затем обновить конфигурацию DbContext, добавив метод `UseChangeTrackingProxies`, как показано в следующем фрагменте кода:

```
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();
optionsBuilder
    .UseChangeTrackingProxies()
    .UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
```

ПРИМЕЧАНИЕ Если вы используете третий подход, то можете отключить часть с `INotifyPropertyChanging`, установив для первого параметра `useChangeTrackingProxies` в методе `UseChangeTrackingProxies` значение `false`. После этого EF Core начнет использовать снимок отслеживания для сравнения.

11.4.3 Использование состояния сущностей в методе SaveChanges

Пока мы узнали, как установить свойство сущности `State` и что `ChangeTracker` можно использовать для отслеживания изменений. Далее мы будем использовать данные свойства `State` в методах `SaveChanges`

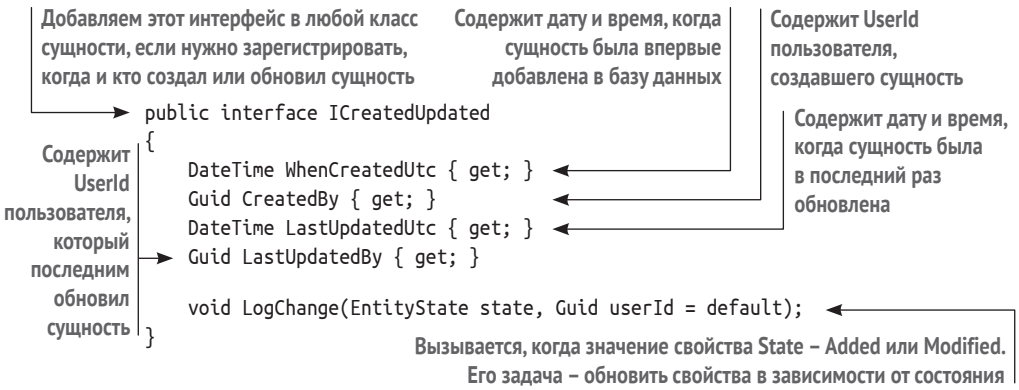
и SaveChangesAsync, чтобы делать некоторые интересные вещи. Вот несколько возможных способов определения того, что должно измениться в базе данных:

- автоматическое добавление дополнительной информации к сущности – например, времени добавления или обновления сущности;
- запись истории изменения данных в базу данных каждый раз, когда изменяется конкретный тип сущности;
- добавление проверки безопасности, чтобы узнать, разрешено ли текущему пользователю обновлять этот конкретный тип сущности.

Базовый подход – переопределить методы SaveChanges и SaveChangesAsync внутри DbContext приложения и выполнить код перед вызовом базового метода SaveChanges или SaveChangesAsync. Мы проверяем значение свойств State перед вызовом базового метода SaveChanges, потому что а) после вызова этого метода свойство State каждой отслеживаемой сущности будет иметь значение Unchanged и б) вам нужно добавить/изменить некоторые сущности, прежде чем они будут записаны в базу данных. Что делать с информацией свойства State, решать вам, но далее приводится пример, который регистрирует, когда сущность была добавлена или обновлена в последний раз, с UserId пользователя, который выполнил эти операции.

В следующем листинге представлен интерфейс, который можно добавить к любому классу сущности. Он определяет свойства, которые вы хотите заполнить при добавлении или обновлении сущности, и метод, который можно использовать, чтобы установить правильные значения этим свойствам.

Листинг 11.6 Интерфейс ICreatedUpdated, определяющий четыре свойства и метод



В следующем листинге показан класс сущности CreatedUpdatedInfo, реализующий интерфейс ICreatedUpdated, который будет обнаружен при вызове измененного метода SaveChanges (см. листинг 11.8). Метод

LogChange, который вызывается в измененном методе SaveChanges, настраивает различные свойства в классе сущности.

Листинг 11.7 Автоматическая установка того, кто и когда обновлял сущность

```
public class CreatedUpdatedInfo : ICreatedUpdated
   {
    public DateTime WhenCreatedUtc { get; private set; }
    public Guid CreatedBy { get; private set; }
    public DateTime LastUpdatedUtc { get; private set; }
    public Guid LastUpdatedBy { get; private set; }

    public void LogChange(EntityEntry entry,
        Guid userId = default)
    {
        if (entry.State != EntityState.Added &&
            entry.State != EntityState.Modified)
            return;

        var timeNow = DateTime.UtcNow;
        LastUpdatedUtc = timeNow;
        LastUpdatedBy = userId;
        if (entry.State == EntityState.Added)
        {
            WhenCreatedUtc = timeNow;
            CreatedBy = userId;
        }
        else
        {
            entry.Property(
                nameof(ICreatedUpdated.LastUpdatedUtc))
                .IsModified = true;
            entry.Property(
                nameof(ICreatedUpdated.LastUpdatedBy))
                .IsModified = true;
        }
    }
}
```

Класс сущности наследуется от интерфейса ICreatedUpdated, а это означает, что любое добавление или обновление сущности регистрируется

У этих свойств закрытые методы записи, поэтому только метод LogChange может их изменять

Его задача – обновлять созданные и обновленные сущности. В качестве параметра передается UserId, если он задан

Этот метод обрабатывает только состояния Added или Modified

Получает текущее время, поэтому время добавления и обновления будет одинаковым при создании

Всегда устанавливает LastUpdatedUtc и LastUpdatedBy

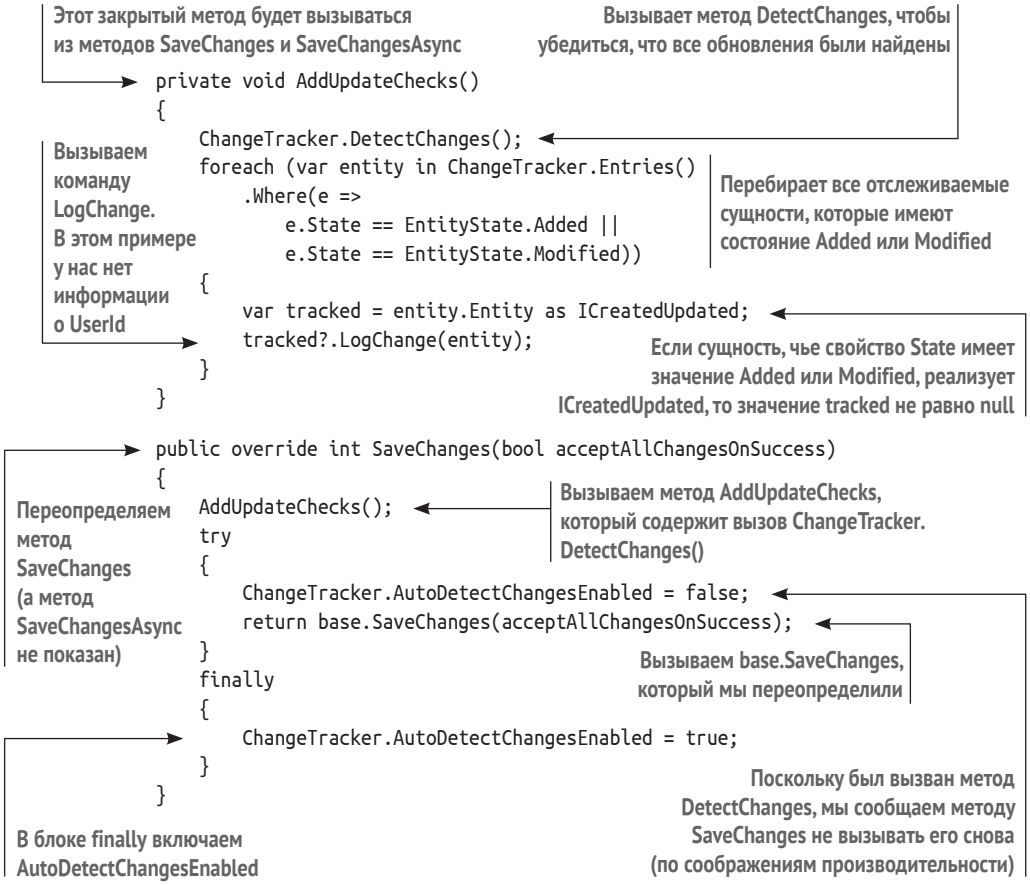
Если это добавление новой сущности, свойства WhenCreatedUtc и CreatedBy обновляются

Из соображений производительности мы отключили метод DetectChanges, поэтому необходимо вручную пометить свойства как измененные

Следующий шаг – переопределить все версии метода SaveChanges внутри DbContext приложения, а затем перед вызовом базового метода SaveChanges вызвать метод AddUpdateChecks, показанный в листинге 11.8. Этот метод ищет сущности с состояниями Added или Modified и реализующие интерфейс ICreatedUpdated. Если он находит сущность (или сущности), соответствующую этим критериям, то вызывает метод сущности LogChange, чтобы установить для двух свойств правильные значения.

В следующем листинге показан DbContext приложения, `Chapter11-DbContext`, реализующий этот код. (Чтобы код был короче, переопределяется только метод `SaveChanges`. Обычно также переопределяется метод `SaveChangesAsync` с двумя параметрами.) Кроме того, обратите внимание: этот код гарантирует, что метод `ChangeTracker.DetectChanges` вызывается только один раз, потому что, как вы уже видели, его выполнение может занимать довольно продолжительное время.

Листинг 11.8 DbContext ищет добавленные или измененные сущности `ICreatedUpdated`




```
private void AddUpdateChecks()
{
    ChangeTracker.DetectChanges();
    foreach (var entity in ChangeTracker.Entries()
        .Where(e =>
            e.State == EntityState.Added ||
            e.State == EntityState.Modified))
    {
        var tracked = entity.Entity as ICreatedUpdated;
        tracked?.LogChange(entity);
    }
}

public override int SaveChanges(bool acceptAllChangesOnSuccess)
{
    AddUpdateChecks();
    try
    {
        ChangeTracker.AutoDetectChangesEnabled = false;
        return base.SaveChanges(acceptAllChangesOnSuccess);
    }
    finally
    {
        ChangeTracker.AutoDetectChangesEnabled = true;
    }
}
```

Это только один из примеров использования `ChangeTracker` для выполнения действий на основе `State` отслеживаемых сущностей, но он задает общий подход. Возможности здесь бесконечны.

ПРИМЕЧАНИЕ В главе 16 у меня есть еще один пример использования свойства `State` определенных сущностей для частичного обновления базы данных при изменении `Book` или связанных с ней сущностей.

11.4.4 Перехват изменений свойства State с использованием события

В EF Core версии 2.1 были добавлены два события: `ChangeTracker.Tracked`, которое возникает, когда сущность начинает отслеживаться, и `ChangeTracker.StateChanged`, которое возникает при изменении свойства `State` уже отслеживаемой сущности. Эта функция обеспечивает эффект, аналогичный вызову `ChangeTracker.Entries()`, но создавая событие, когда что-то меняется. События `ChangeTracker` полезны для таких функций, как журналирование изменений или выполнение действий при изменении свойства `State` конкретного типа сущности. Но для начала познакомимся с основами.

Более простое событие `Tracked` запускается, когда класс сущности начинает отслеживаться и сообщает, поступила ли сущность из запроса, через свойство `FromQuery`. Такое событие может произойти, когда вы выполняете запрос EF Core (без метода `AsNoTracking`) или начинаете отслеживать класс сущности с помощью методов `Add` или `Attach`. Следующий листинг – это модульный тест, который перехватывает событие `Tracked` при добавлении класса сущности в контекст.

Листинг 11.9 Событие `ChangeTracker.Tracked` и его содержимое

```
Ведет журнал событий Tracked
var logs = new List<EntityTrackedEventArgs>();
context.ChangeTracker.Tracked += delegate(
          object sender, EntityTrackedEventArgs args)
{
          logs.Add(args);
};

//ПОПЫТКА
var entity = new MyEntity {MyString = "Test"};
context.Add(entity);

//ПРОВЕРКА
logs.Count.ShouldEqual(1);
logs.Single().FromQuery.ShouldBeFalse();
logs.Single().Entry.Entity.ShouldEqual(entity);
logs.Single().Entry.State
    .ShouldEqual(EntityState.Added);
```

Регистрируем обработчик событий для события ChangeTracker.Tracked

Этот обработчик событий просто записывает в журнал EntityTrackedEventArgs

Создает класс сущности

Добавляет этот класс сущности в контекст

Содержит одно событие

Сущность не отслеживалась, т. к. получена не из запроса

Вы можете получить доступ к сущности, которая инициировала событие

Кроме того, можно получить текущее свойство State этой сущности

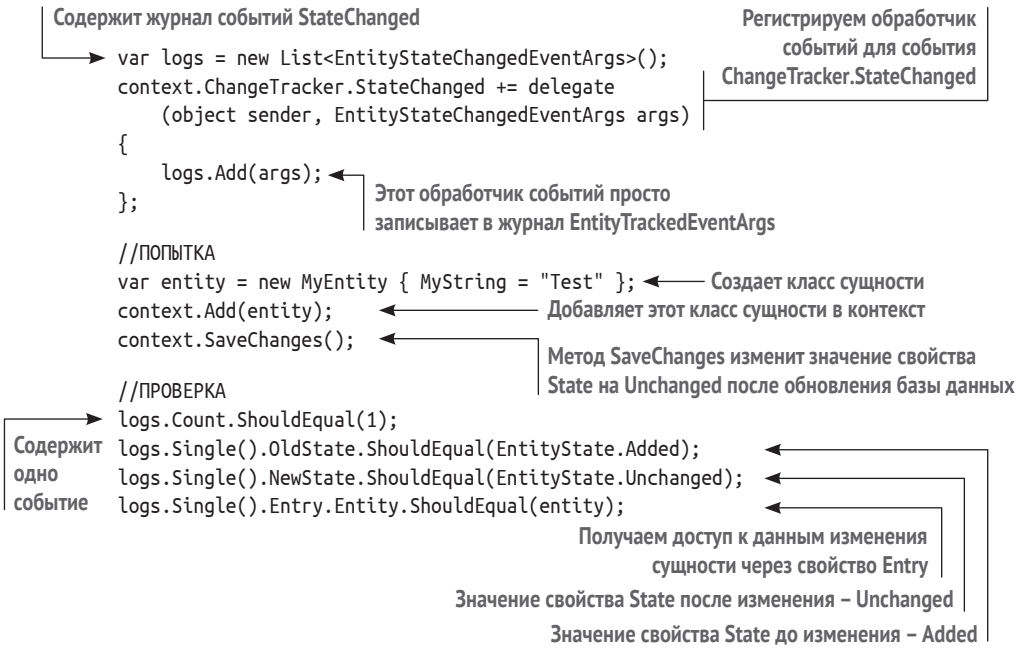
В этом листинге показано, какая информация доступна в данных события. Что касается события `Tracked`, то вы получите параметр `FromQuery`, который будет иметь значение `true`, если запрос отслеживался. Свойство `Entry` дает информацию о сущности.

В этом примере следует отметить, что метод `context.Add(entity)` запускает событие `Tracked`, но не событие `StateChanged`. Если вы хо-

тите обнаружить только что добавленный класс сущности, то сделать это можно, используя лишь событие `Tracked`.

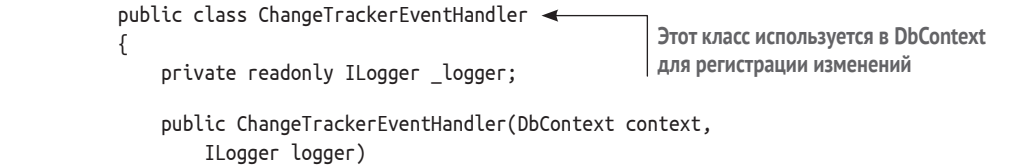
Событие `StateChanged` похоже, но содержит другую информацию. Следующий листинг перехватывает событие `StateChanged` при вызове метода `SaveChanges`. Событие содержит значение свойства `State` сущности до вызова этого метода в свойстве `OldState` и значение после вызова в свойстве `NewState`.

Листинг 11.10 Событие `ChangeTracker.StateChanged` и его содержимое



В листинге показано, что вы получаете значение свойства `State` до и после изменения, используя свойства `OldState` и `NewState` соответственно. Теперь, когда вы познакомились с событиями `ChangeTracker`, воспользуемся ими для журналирования изменений в каком-либо другом хранилище. Но в следующем листинге я показываю класс, который запишет два события `ChangeTracker` в журнал, используя интерфейс NET `ILogger`.

Листинг 11.11 Класс, содержащий код для записи событий `ChangeTracker` в журнал



```
    {
        _logger = logger;
        context.ChangeTracker.Tracked += TrackedHandler;
        context.ChangeTracker.StateChanged += StateChangeHandler;
    }

    private void TrackedHandler(object sender,
        EntityTrackedEventArgs args)
    {
        if (args.FromQuery)
            return;

        var message = $"Entity: {NameAndPk(args.Entry)}. " +
            $"Was {args.Entry.State}";
        _logger.LogInformation(message);
    }

    private void StateChangeHandler(object sender,
        EntityStateChangedEventArgs args)
    {
        var message = $"Entity: {NameAndPk(args.Entry)}. " +
            $"Was {args.OldState} and went to {args.NewState}";
        _logger.LogInformation(message);
    }
}
```

Будем вести журнал через ILogger

Добавляет обработчик событий Tracked

Добавляет обработчик событий StateChanged

Обрабатывает события Tracked

Мы не хотим регистрировать сущности, которые были получены из запроса

Формирует полезное сообщение при Add или Attach

Обработчик событий StateChanged регистрирует любые изменения

Теперь добавьте этот код в конструктор DbContext приложения, как показано в следующем листинге.

Листинг 11.12 Добавление `ChangeTrackerEventHandler` в DbContext приложения

```
public class Chapter11DbContext : DbContext
{
    private ChangeTrackerEventHandler _trackerEventHandler;

    public Chapter11DbContext(
        DbContextOptions<Chapter11DbContext> options,
        ILogger logger = null)
        : base(options)
    {
        if (logger != null)
            _trackerEventHandler = new
                ChangeTrackerEventHandler(this, logger);
    }
    //… Остальной код не указан;
}
```

DbContext приложения, для которого вы хотите регистрировать изменения

Нам нужен экземпляр обработчика событий, пока существует DbContext

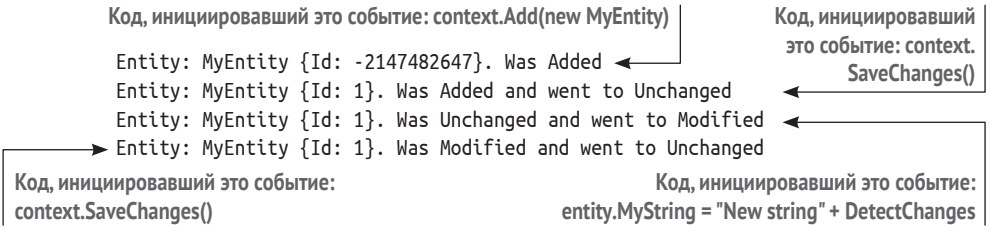
Добавляем ILogger в конструктор

Если ILogger доступен, регистрируем обработчики

Создает класс обработчика событий, который регистрирует обработчики

Это простой пример, но он показывает, насколько эффективны события `ChangeTracker`. Мои сообщения журнала довольно просты (см. следующий листинг), но можно легко их расширить, чтобы увидеть подробное описание того, какие свойства были изменены, включая `UserId` пользователя, который вносил изменения, и т. д.

Листинг 11.13 Пример вывода журналирования событий ChangeTrackerEventHandler

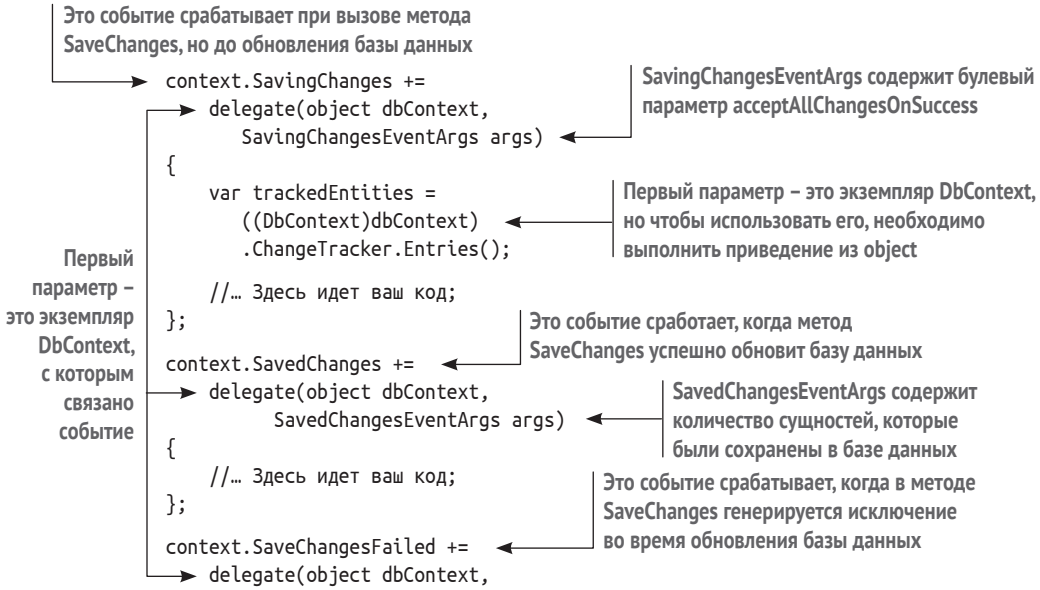


11.4.5 Запуск событий при вызове методов SaveChanges и SaveChangesAsync

EF Core 5 представил события `SavingChanges`, `SavedChanges` и `SaveChangesFailed`, которые вызываются соответственно перед сохранением данных в базе, после того как данные были успешно сохранены, или если сохранение окончилось неудачей. Эти события позволяют узнать, что происходит в методах `SaveChanges` и `SaveChangesAsync`. Можно использовать эти события для регистрации того, что было записано в базу данных, или предупредить кого-либо, если внутри этих методов генерируется какое-то исключение.

Чтобы использовать эти события, необходимо подписаться на события `SavingChanges` и `SavedChanges`. В следующем листинге показано, как это сделать.

Листинг 11.14 Как подписаться на события SavingChanges и SavedChanges



```
        SaveChangesFailedEventArgs args)
{
    //… Здесь идет ваш код;
};
```

SavingChangesEventArgs содержит исключение, которое произошло во время обновления базы данных

Чтобы использовать эти события, нужно знать о них несколько вещей:

- как и в случае со всеми событиями C#, подписка на них длится только до тех пор, пока существует экземпляр DbContext;
- события вызываются методами `SaveChanges` и `SaveChangesAsync`;
- событие `SavingChanges` вызывается перед вызовом метода `ChangeTracker.DetectChanges`, поэтому если вы хотите реализовать код, показанный в разделе 11.4.3, чтобы обновить сущности, используя их свойство `State`, сначала необходимо вызвать метод `ChangeTracker.DetectChanges`. Однако такой подход – не лучшая идея, потому что метод `DetectChanges` будет вызываться дважды, а это может вызвать проблемы с производительностью.

11.4.6 Перехватчики EF Core

В EF Core 3.0 были представлены перехватчики, позволяющие перехватывать, изменять и/или подавлять операции EF Core, включая низкоуровневые операции с базой данных, например выполнение команды, а также операции более высокого уровня, например вызовы метода `SaveChanges`. Эти перехватчики обладают мощными возможностями. Например, они могут изменять команды, отправляемые в базу данных.

Это расширенная функция, поэтому в данном разделе просто указывается, что она доступна. Кроме того, существует неплохая документация EF Core по перехватчикам. Она содержит много полезных примеров и занимает около 15 страниц. Дополнительную информацию можно найти на странице http://mng.bz/zGJQ.

11.5 Использование команд SQL в приложении EF Core

В EF Core есть методы, позволяющие использовать команды SQL как часть LINQ-запроса или записи в базу данных, например SQL UPDATE. Эти команды полезны, когда запрос, который вы хотите выполнить, нельзя выразить с помощью LINQ, например когда он вызывает хранимую процедуру SQL, или когда запрос LINQ приводит к неэффективному коду SQL, который отправляется в базу данных.

ОПРЕДЕЛЕНИЕ Хранимая процедура SQL – это набор команд SQL (с параметрами или без), которые можно выполнить. Обыч-

но эти команды выполняют чтение и/или запись в базу данных. Набор команд SQL хранится в базе данных как хранимая процедура с именем. В таком случае хранимую процедуру можно вызвать как часть команды SQL.

Команды SQL в EF Core спроектированы для обнаружения атак с использованием SQL-инъекций – атак, при которых злоумышленник заменяет, скажем, значение первичного ключа некими командами SQL, которые извлекают дополнительные данные из базы данных. EF Core предоставляет два типа команд SQL:

- методы, названия которых оканчиваются на слово `Raw`, например `FromSqlRaw`. В этих командах указываются отдельные параметры, которые проверяются;
- методы, названия которых оканчиваются на слово `Interpolated`, например `FromSqlInterpolated`. В качестве параметра эти методы принимают строку, использующую строковую интерполяцию C# 6 с параметрами в строке, например `$"SELECT * FROM Books WHERE BookId = {myKey}"`. EF Core может проверять каждый параметр в интерполированном строковом типе.

ПРЕДУПРЕЖДЕНИЕ Если вы создаете интерполированную строку вне команды, например `var badSQL = $"SELECT … WHERE BookId = {myKey}"-`, а затем используете ее в такой команде, как `FromSqlRaw(badSQL)`, EF Core не сможет предотвратить атаки с использованием SQL-инъекций. Нужно использовать метод `FromSqlRaw` с параметрами или метод `FromSqlInterpolated` с параметрами, встроенными в строковую интерполяцию.

Можно включать команды SQL в команды EF несколькими способами. Помимо показа каждой из групп, я буду использовать в примерах смесь синхронных версий `…Raw` и `…Interpolated`. У каждого показываемого мною метода есть асинхронная версия, кроме метода `GetDbConnection`. Рассматриваются следующие группы команд:

- синхронные и асинхронные методы `FromSqlRaw/FromSqlInterpolated`, позволяющие использовать команду SQL в запросе EF Core;
- синхронные и асинхронные методы `ExecuteSqlRaw/ExecuteSqlInterpolated`, выполняющие команду без запроса результата;
- метод Fluent API `ToSqlQuery`, отображающий класс сущности в SQL-запрос;
- команда `Reload/ReloadAsync`, используемая для обновления загруженной EF Core сущности, которая была изменена методом `ExecuteSql…`;
- метод EF Core `GetDbConnection`, обеспечивающий низкоуровневый доступ к библиотекам баз данных для прямого доступа к базе.

EF6 Команды в EF Core для SQL-доступа к базе данных отличаются от способа, который используется в EF6.x.

11.5.1 Методы FromSqlRaw/FromSqlInterpolated: использование SQL в запросе EF Core

Методы `FromSqlRaw/FromSqlInterpolated` позволяют добавлять команды SQL в стандартный запрос EF Core, включая команды, которые нельзя вызвать из EF Core, например хранимые процедуры. Вот пример вызова хранимой процедуры, которая возвращает только книги, получившие среднюю оценку, равную переданному значению.

Листинг 11.15 Использование метода `FromSqlInterpolated` для вызова хранимой процедуры SQL

Вы выполняете запрос обычным способом, используя DbSet<T>, который хотите прочитать

Метод FromSqlInterpolated позволяет вставить команду SQL

Использует функцию интерполяции строк C# 6 для предоставления параметра

Необходимо удалить все фильтры запросов; в противном случае SQL будет недействительным

```
int filterBy = 5;
var books = context.Books
    .FromSqlInterpolated(
        $"EXECUTE dbo.FilterOnReviewRank @RankFilter = {filterBy}")
    .IgnoreQueryFilters()
    .ToList();
```

Есть несколько правил относительно SQL-запроса:

- запрос должен возвращать данные для всех свойств типа сущности (но есть способ обойти это правило; см. раздел 11.5.5);
- имена столбцов в наборе результатов должны совпадать с именами столбцов, в которые отображаются свойства;
- SQL-запрос не может содержать связанные данные, но можно добавить метод `Include` для загрузки связанных навигационных свойств (см. листинг 11.16).

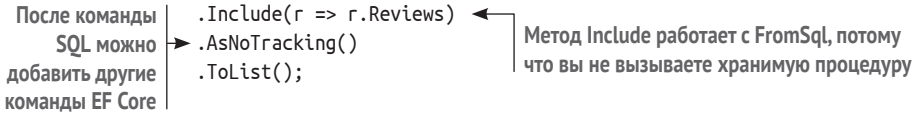
После команды SQL можно добавить другие команды EF Core, например `Include`, `Where` и `OrderBy`. В следующем листинге показана команда SQL, фильтрующая результаты по средней оценке с методом `Include` и командой `AsNoTracking`.

Листинг 11.16 Пример добавления дополнительных команд EF Core в конец SQL-запроса

SQL вычисляет среднюю оценку и использует ее в операторе WHERE

В этом случае используется обычный метод проверки и подстановки параметров SQL – {0}, {1}, {2} и т. д.

```
double minStars = 4;
var books = context.Books
    .FromSqlRaw(
      "SELECT * FROM Books b WHERE " +
        "(SELECT AVG(CAST([NumStars] AS float)) " +
        "FROM dbo.Review AS r " +
         "WHERE b.BookId = r.BookId) >= {0}", minStars)
```

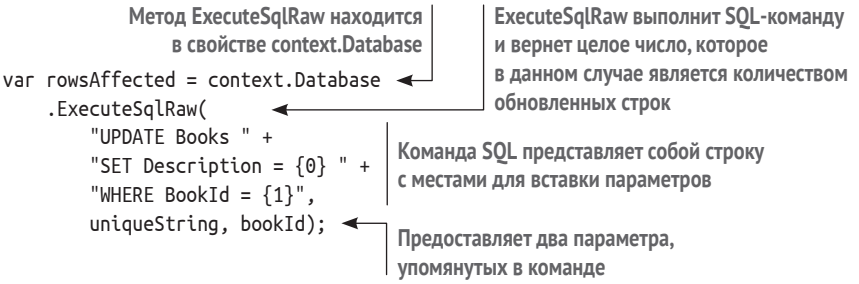


ПРЕДУПРЕЖДЕНИЕ Если вы используете фильтры запросов на уровне модели (см. раздел 6.1.7), SQL-код, который вы можете написать, имеет ограничения. Например, оператор `ORDER BY` не сработает. Чтобы обойти эту проблему, можно применить метод `IgnoreQueryFilters` после команды SQL и воссоздать фильтр запросов на уровне модели в коде SQL.

11.5.2 Методы ExecuteSqlRaw и ExecuteSqlInterpolated: выполнение команды без получения результата

Помимо помещения в запрос команд SQL, можно выполнять команды SQL без получения результата с помощью методов EF Core `ExecuteSqlRaw` и `ExecuteSqlInterpolated`. Как правило, это команды `UPDATE` и `DELETE`, но можно вызвать любую команду SQL без запроса результата. В следующем листинге показана команда `UPDATE`, принимающая два параметра.

Листинг 11.17 Метод `ExecuteSqlCommand`, выполняющий команду `UPDATE`



Метод `ExecuteSqlRaw` возвращает целое число, которое полезно для проверки, что команда была выполнена так, как мы и ожидали. В этом примере мы ожидаем, что метод вернет 1, чтобы показать, что он нашел и обновил строку в таблице Books, у которой был предоставленный вами первичный ключ.

11.5.3 Метод Fluent API ToSqlQuery: отображение классов сущностей в запросы

EF Core 5 предоставил способ отображать класс сущности в SQL-запрос с помощью метода Fluent API `ToSqlQuery`. Эта функция позволяет скрыть код SQL внутри конфигурации DbContext приложения, а разработчики могут использовать свойство `DbSet<T>` в запросах, как если

бы это был обычный класс сущности, отображаемый в сущность. Конечно, это класс сущности с доступом только на чтение. Если вам нужна версия с доступом на чтение и запись, см. следующее примечание.

> **ПРИМЕЧАНИЕ** В EF Core 5 добавлена возможность настройки отображения класса сущности в таблицу (для создания, обновления и удаления) и представление (для чтения): http://mng.bz/0rY6.

В качестве примера мы создадим класс сущности `BookSqlQuery`, возвращающий три значения для класса сущности `Book`: `BookId`, `Title` и среднюю оценку этой книги в свойстве `AverageVotes`. Этот класс показан в следующем листинге.

Листинг 11.18 Класс `BookSqlQuery` для отображения в SQL-запрос

```
public class BookSqlQuery
{
    [Key]
    public int BookId { get; set; }

    public string Title { get; set; }

    public double? AverageVotes { get; set; }
}
```

Первичный ключ возвращаемой книги

Название книги

Средняя оценка этой книги на основе свойства Review, NumStars

Теперь нужно настроить этот класс сущности для SQL-запроса, используя метод Fluent API `ToSqlQuery`, как показано в следующем листинге.

Листинг 11.19 Настройка класса сущности `BookSqlQuery` для SQL-запроса

```
public class BookDbContext : DbContext
{
    //… Остальные DbSet удалены для ясности;
    public DbSet<BookSqlQuery> BookSqlQueries { get; set; }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        //… Остальные конфигурации удалены для ясности;

        modelBuilder.Entity<BookSqlQuery>().ToSqlQuery(
        @"SELECT BookId
            ,Title
            ,(SELECT AVG(CAST([r0].[NumStars] AS float))
        FROM Review AS r0
        WHERE t.BookId = r0.BookId) AS AverageVotes
        FROM Books AS t");
    }
}
```

Добавляем свойство DbSet<T> для класса сущности BookSqlQuery, чтобы упростить выполнение запросов

Метод ToSqlQuery отображает класс сущности в SQL-запрос

Возвращает три значения для каждой книги

Можно добавить команды LINQ, например `Where` и `OrderBy`, обычным способом, но возвращаемые данные подчиняются тем же правилам, что и методы `FromSqlRaw` и `FromSqlInterpolated` (раздел 11.5.1).

11.5.4 Метод Reload: используется после команд ExecuteSql

Если у вас уже есть загруженная сущность (и она отслеживается) и вы используете методы `ExecuteSqlRaw` или `ExecuteSqlInterpolated` для изменения данных в базе, то отслеживаемая сущность становится неактуальной. Эта ситуация может вызвать проблемы позже, потому что EF Core не знает, что значения были изменены. Чтобы решить эту проблему, в EF Core есть метод `Reload/ReloadAsync`, который обновляет сущность, отправляя повторный запрос к базе данных.

В следующем листинге мы загружаем сущность, изменяем ее содержимое с помощью метода `ExecuteSqlCommand`, а затем используем метод `Reload`, чтобы убедиться, что содержимое сущности совпадает с тем, что есть в базе данных.

Листинг 11.20 Использование метода Reload для обновления содержимого существующей сущности

```
var entity = context.Books.
    Single(x => x.Title == "Quantum Networking");
var uniqueString = Guid.NewGuid().ToString();

context.Database.ExecuteSqlRaw(
        "UPDATE Books " +
        "SET Description = {0} " +
        "WHERE BookId = {1}",
        uniqueString, entity.BookId);

context.Entry(entity).Reload();
```

Загружает сущность Book обычным способом

Использует метод ExecuteSqlRaw для изменения столбца Description той же сущности Book

При вызове метода Reload EF Core повторно считывает эту сущность, чтобы убедиться, что локальная копия актуальна

В конце этого кода экземпляр сущности будет соответствовать тому, что находится в базе данных.

11.5.5 GetDbConnection: выполнение собственных команд SQL

Если EF Core не может предоставить нужные вам функции запросов, нужно вернуться к другому методу доступа к базе данных, который может это сделать. Для некоторых низкоуровневых библиотек для работы с базами данных требуется писать намного больше кода, но это позволяет обеспечить более прямой доступ к базе данных, чтобы вы могли делать почти все, что нужно. Обычно эти низкоуровневые библиотеки баз данных предназначены для конкретного сервера. В этом разделе используется библиотека NuGet, Dapper (https://github.com/StackExchange/Dapper). Dapper – это простой инструмент отображе-

ния объектов для .NET, который иногда называют микро-ORM. Dapper простой, но быстрый. Он использует библиотеку ADO.NET для доступа к базе данных и добавляет автокопирование столбцов в свойства класса.

В следующем листинге Dapper используется для чтения определенных столбцов в класс, не являющийся сущностью, `RawSqlDto`, у которого есть свойства `BookId`, `Title` и `AverageVotes`, поэтому вы можете загружать только те столбцы, которые вам нужны. В этом примере Dapper используется для запроса к той же базе данных, с которой связан DbContext вашего приложения. Запрос Dapper возвращает один класс `RawSqlDto` с данными в трех свойствах для строки Books, где столбец `BookId` (первичный ключ) имеет значение 4.

Листинг 11.21 Получение `DbConnection` от EF Core для выполнения SQL-запроса через Dapper

```
var connection = context.Database.GetDbConnection();
string query = "SELECT b.BookId, b.Title, " +
               "(SELECT AVG(CAST([NumStars] AS float)) " +
               "FROM dbo.Review AS r " +
               "WHERE b.BookId = r.BookId) AS AverageVotes " +
               "FROM Books b " +
               "WHERE b.BookId = @bookId";

var bookDto = connection
    .Query<RawSqlDto>(query, new
    {
        bookId = 4
    })
    .Single();
```

Получает подключение DbConnection к базе данных, которое может использовать Dapper

Создает SQL-запрос, который вы хотите выполнить

Вызывает метод Dapper Query с типом возвращаемых данных

Предоставляет параметры Dapper для добавления в SQL-запрос

СОВЕТ ПО ПРОИЗВОДИТЕЛЬНОСТИ Методы `FromSqlRaw` и `FromSqlInterpolated` должны возвращать все столбцы, отображаемые в вызовы сущностей, и даже если вы добавите метод LINQ `Select` после методов `FromSqlRaw` или `FromSqlInterpolated`, они все равно вернут все столбцы. В результате Dapper, вероятно, будет быстрее загружать несколько столбцов из базы данных, чем любой из методов EF Core `FromSql`....

Не бойтесь сочетать EF Core и Dapper, особенно если у вас есть проблемы с производительностью. Я использую Dapper с EF Core в третьей части, чтобы повысить производительность, потому что я написал улучшенный SQL-запрос, который выполнял сортировку по средним оценкам. Обратная сторона использования Dapper состоит в том, что он ничего не знает о навигационных свойствах, поэтому для работы со связанными сущностями в Dapper требуется больше кода, чем в EF Core.

11.6 Доступ к информации о классах сущностей и таблицам базы данных

Иногда полезно получить информацию о том, как классы и свойства сущностей отображаются в таблицы и столбцы базы данных. EF Core предоставляет два источника информации, один делает акцент на классах сущностей, а другой больше фокусируется на базе данных:

- `context.Entry(entity).Metadata` – имеет более 20 свойств и методов, которые предоставляют информацию о первичном и внешнем ключах и навигационных свойствах;
- `context.Model` – имеет набор свойств и методов, обеспечивающий аналогичный набор данных для свойства `Metadata`, но больше внимания уделяет таблицам базы данных, столбцам, ограничениям, индексам и т. д.

Вот несколько примеров того, как можно использовать эту информацию для автоматизации определенных служб:

- рекурсивное посещение класса сущности и его связей, чтобы можно было применить некое действие в каждом классе сущности, например сбросить значения первичного ключа;
- получение настроек класса сущности, таких как его поведение при удалении;
- поиск имени таблицы и имен столбцов, используемых классом сущности, чтобы можно было создавать SQL-команды с правильными именами таблиц и столбцов.

> **EF6** EF6.x предоставлял некоторую информацию о модели, но ее было сложно использовать, и она была неполной. В EF Core есть исчерпывающий и простой в использовании набор сведений о модели, но документации, кроме комментариев к методам, не так много.

В следующих разделах приведены примеры использования этих источников.

11.6.1 Использование context.Entry(entity).Metadata для сброса первичных ключей

В разделе 6.2.3 вы узнали, как скопировать класс сущности с определенными связями с помощью сброса первичных ключей вручную. Мне нужна была аналогичная функция для клиентского приложения, поэтому я создал службу, которая автоматически сбрасывает первичные ключи. Она может послужить неплохим примером использования `context.Entry(entity).Metadata`.

Пример из раздела 6.2.3 копировал сущность `Order` с двумя сущностями `LineItem`, но не нужно было копировать класс сущности

Book. Следующий листинг представляет собой копию листинга из раздела 6.2.3.

Листинг 11.22 Создание заказа с двумя сущностями LineItem, готовыми к копированию

```
var books = context.SeedDatabaseFourBooks();
var order = new Order
{
    CustomerId = Guid.Empty,
    LineItems = new List<LineItem>
    {
        new LineItem
        {
            LineNum = 1, ChosenBook = books[0], NumBooks = 1
        },
        new LineItem
        {
            LineNum = 2, ChosenBook = books[1], NumBooks = 2
        },
    }
};
context.Add(order);
context.SaveChanges();
```

Создаем Order с двумя LinItem, которые вы хотите скопировать

Для этого теста мы добавляем четыре книги, которые будут использоваться в качестве тестовых данных

Устанавливаем для CustomerId значение по умолчанию, чтобы фильтр запроса позволил прочитать заказ обратно

Добавляет первую сущность LineNum, связанную с первой книгой

Добавляет вторую сущность LineNum, связанную со второй книгой

Записывает эту сущность Order в базу данных

В главе 6 мы читаем классы сущностей Order и LineItems, а затем сбрасываем первичные ключи вручную. А в следующем примере мы создаем класс PkResetter для автоматического выполнения этой задачи. Этот код показан в следующем листинге.

Листинг 11.23 Использование метаданных для посещения каждой сущности и сброса ее первичного ключа

```
public class PkResetter
{
    private readonly DbContext _context;
    private readonly HashSet<object> _stopCircularLook;

    public PkResetter(DbContext context)
    {
        _context = context;
        _stopCircularLook = new HashSet<object>();
    }

    public void ResetPksEntityAndRelationships(object entityToReset)
    {
        if (_stopCircularLook.Contains(entityToReset))
            return;

        _stopCircularLook.Add(entityToReset);
```

Используется для остановки рекурсии

Этот метод рекурсивно просматривает все связанные сущности и сбрасывает их первичные ключи

Запоминает, что этот метод посетил эту сущность

Если метод уже просматривал эту сущность, он завершает работу

```
var entry = _context.Entry(entityToReset);
if (entry == null)
    return;

var primaryKey = entry.Metadata.FindPrimaryKey();
if (primaryKey != null)
{
    foreach (var primaryKeyProperty in primaryKey.Properties)
    {
        primaryKeyProperty.PropertyInfo
            .SetValue(entityToReset,
            GetDefaultValue(
                primaryKeyProperty.PropertyInfo.PropertyType));
    }
}

foreach (var navigation in entry.Metadata.GetNavigations())
{
    var navProp = navigation.PropertyInfo;

    var navValue = navProp.GetValue(entityToReset);
    if (navValue == null)
        continue;

    if (navigation.IsCollection)
    {
        foreach (var item in (IEnumerable)navValue)
        {
            ResetPksEntityAndRelationships(item);
        }
    }
    else
    {
       ResetPksEntityAndRelationships(navValue);
    }
}
    }
}
```

Обрабатывает сущность, неизвестную вашей конфигурации

Получает информацию о первичном ключе для этой сущности

Сбрасывает каждое свойство, используемое в первичном ключе, до значения по умолчанию

Получает все навигационные свойства для этой сущности

Получает свойство, содержащее навигационное свойство

Получает значение навигационного свойства

Если null, пропускает навигационное свойство

Если навигационное свойство – это коллекция, то посещает все сущности

Рекурсивно посещает каждую сущность в коллекции

Если это единичный объект, то посещает эту сущность

Может показаться, что здесь слишком много кода для сброса трех первичных ключей, но он будет работать с любой конфигурацией класса сущности, поэтому можно использовать его где угодно. Вот список различных свойств и методов `Metadata`, использованных в листинге 11.23:

- *найти первичный ключ сущности* – `entry.Metadata.FindPrimaryKey();`
- *получить свойства первичного ключа* – `primaryKeyProperty.PropertyInfo;`
- *найти навигационные связи сущности* – `Metadata.GetNavigations();`
- *получить свойство навигационной связи* – `navigation.PropertyInfo;`

- *проверить, является ли навигационное свойство коллекцией,* – `navigation.IsCollection`.

ПРИМЕЧАНИЕ Класс `PkResetter` предполагает, что первичные ключи и навигационные свойства хранятся в свойстве, но на самом деле эти значения могут находиться в резервных полях или теневых свойствах. Это упрощение было использовано для того, чтобы код был короче и его легче было читать.

11.6.2 Использование свойства context.Model для получения информации о базе данных

Свойство `context.Model` дает доступ к модели базы данных, которую EF Core создает при первом использовании DbContext приложения. Модель содержит некоторые данные, похожие на `context.Entry(entity).Metadata`, но также содержит конкретную информацию о схеме базы данных. Поэтому если вы хотите что-то сделать на стороне базы данных, то `context.Model` – это верный источник информации, который следует использовать.

Я использовал исходное свойство `context.Model` для создания библиотеки EfCore.EfSchemaCompare, о которой упоминал в разделе 9.5.3. Но в качестве небольшого примера мы создадим метод, возвращающий SQL-команду для удаления набора сущностей с общим внешним ключом, чтобы улучшить производительность удаления группы зависимых сущностей.

Если вы удаляете группу зависимых сущностей через EF Core, то обычно читаете все сущности, которые нужно удалить, а EF Core удаляет каждую сущность с помощью отдельной команды SQL. Метод из следующего листинга создает одну SQL-команду, которая удаляет все зависимые сущности в одной команде без необходимости читать их. Поэтому процесс идет намного быстрее, чем в EF Core, особенно с большими коллекциями.

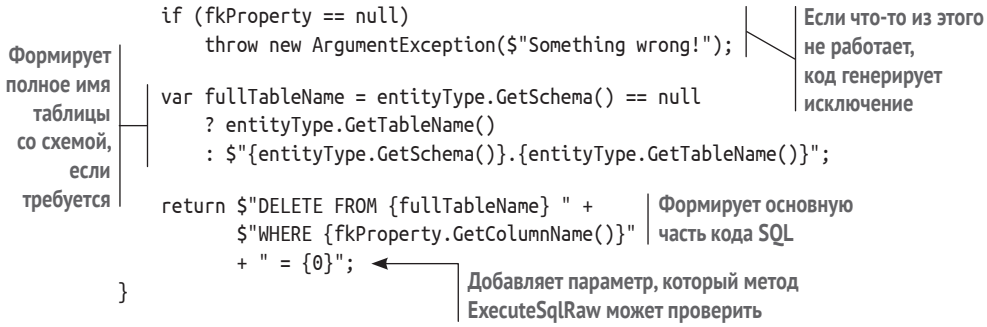
Листинг 11.24 Использование свойства `context.Model` для более быстрого удаления зависимой сущности

Этот метод позволяет быстро удалить все сущности, связанные с основной сущностью

```
public string BuildDeleteEntitySql<TEntity>
    (DbContext context, string foreignKeyName)
    where TEntity : class
{
    var entityType = context.Model.FindEntityType(typeof(TEntity));
    var fkProperty = entityType?.GetForeignKeys()
        .SingleOrDefault(x => x.Properties.Count == 1
            && x.Properties.Single().Name == foreignKeyName)
        ?.Properties.Single();
```

Получает информацию о модели для данного типа или null, если тип не отображается в базу данных

Ищет внешний ключ с единственным свойством с заданным именем



Найдя нужную сущность или таблицу и проверив соответствие имени внешнего ключа, можно создать SQL-код. Как видно из листинга, у нас есть доступ к имени таблицы и схеме плюс имя столбца внешнего ключа. Следующий фрагмент кода показывает вывод метода `BuildDeleteEntitySql` из листинга 11.24 с классом сущности `Review` для `TEntity` и имя внешнего ключа `BookId`:

```
DELETE FROM Review WHERE BookId = {0}
```

Команда SQL применяется к базе данных путем вызова метода `ExecuteSqlRaw`, со строкой SQL в качестве первого параметра и значением внешнего ключа в качестве второго параметра.

> **ПРИМЕЧАНИЕ** Класс `BuildDeleteEntitySql` предполагает, что внешний ключ является единственным, но он может быть составным ключом с несколькими значениями. Такое упрощение было использовано для того, чтобы код был короче и его легче было читать.

Хотя это и простой пример, он показывает, что использование методов `Model` позволяет получить информационные классы сущностей с их связями и отобразить эти классы в схему базы данных.

11.7 Динамическое изменение строки подключения DbContext

EF Core 5 упрощает изменение строки подключения в экземпляре DbContext приложения. Теперь он предоставляет метод `SetConnectionString`, позволяющий изменить строку подключения в любое время, чтобы вы могли изменить базу данных, к которой обращаетесь. Обычно я использую эту функцию для выбора разных баз данных на основе лица, выполнившего вход, того, где находится пользователь, и т. д. Этот процесс известен как *сегментирование базы данных*. Он обеспечивает более высокую производительность, поскольку данные пользователей распределены по нескольким базам данных. Кроме

того, он может повысить безопасность, разместив все данные для одной группы пользователей в одной базе данных. На рис. 11.6 показан метод `SetConnectionString`, используемый для реализации системы сегментирования базы данных с помощью EF Core.

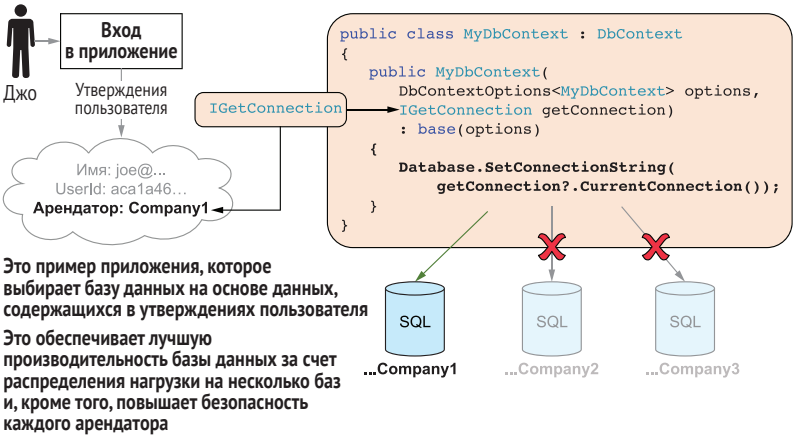


Рис. 11.6 Пользователь входит в приложение ASP.NET Core. Приложение использует данные пользователя, чтобы определить, к какой базе данных следует получить доступ, и добавляет утверждение `Tenant` для хранения этой информации. Такой вариант работает, потому что при создании DbContext приложения выполняется код конструктора для настройки строки подключения к базе данных. Этот код использует внедренный класс `IGetConnection`, возвращающий строку подключения на основе утверждения `Tenant`, которое связывает пользователя с правильной базой данных

В EF Core 5 было внесено еще одно важное изменение: строка подключения может иметь значение `null`, когда вы впервые создаете DbContext приложения. (До EF Core 5 строка подключения не могла иметь значение `null`.) Строка подключения может иметь это значение, пока вам не понадобится доступ к базе данных. Это полезная функция, потому что при запуске не будет информации о пользователе, поэтому строка подключения будет иметь значение `null`. Однако после этого изменения, появившегося EF Core 5, ваш код конфигурации может работать без строки подключения. Еще один пример – моя библиотека EfCore.GenericServices, которой необходимо сканировать сущности, используемые в базе данных при запуске. Теперь эта библиотека работает, даже если строка подключения имеет значение `null`.

11.8 Решение проблем, связанных с подключением к базе данных

При работе с серверами реляционных баз данных, особенно в облаке, запрос к базе данных может завершиться ошибкой из-за истече-

ния времени ожидания подключения или возникновения случайных ошибок. В EF Core есть функция стратегии выполнения, позволяющая определить, что должно произойти при истечении времени ожидания, сколько тайм-аутов разрешено и т. д. Обеспечение стратегии выполнения может снизить вероятность сбоя приложения из-за проблем с подключением или случайных внутренних ошибок.

> **EF6** Стратегия выполнения EF Core – это улучшение аналогичной стратегии в EF6.x, поскольку EF Core может самостоятельно обрабатывать повторные попытки транзакции.

Поставщик базы данных SQL Server включает стратегию выполнения, которая специально адаптирована для SQL Server (включая SQL Azure). Он знает о типах исключений, при которых можно осуществить повторную попытку, и у него есть разумные настройки по умолчанию для максимального числа повторных попыток, задержки между повторными попытками и т. д. В этом листинге показано, как применить эту стратегию к настройке SQL Server. Стратегия выполнения выделена жирным шрифтом.

Листинг 11.25 Настройка DbContext со стандартной стратегией выполнения SQL

```
var connection = @"Server=(localdb)\mssqllocaldb;Database=… etc.";
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();

optionsBuilder.UseSqlServer(connection,
    option => option.EnableRetryOnFailure());
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{
  … Обычный код для использования контекста;
```

Обычные запросы EF Core или вызовы метода `SaveChanges` будут автоматически повторяться без каких-либо действий с вашей стороны. Каждый запрос и каждый вызов метода `SaveChanges` повторяются как единое целое, если происходит кратковременный отказ. Но обеспечение корректной работы транзакций с использованием стратегий выполнения требует дополнительной настройки.

11.8.1 Обработка транзакций базы данных с использованием стратегии выполнения

Из-за логики работы стратегии выполнения необходимо адаптировать любой код, использующий транзакции базы данных, в которых есть несколько вызовов метода `SaveChanges` внутри транзакции. (См. раздел 4.7.2 для получения информации о том, как работают

транзакции.) Стратегия выполнения работает путем отката всей транзакции в случае временного сбоя и последующего воспроизведения каждой операции в транзакции; каждый запрос и каждый вызов метода `SaveChanges` повторяются как единое целое. Чтобы все операции в транзакции повторялись, стратегия выполнения должна контролировать код транзакции.

В следующем листинге показано добавление стратегии выполнения SQL Server `EnableRetryOnFailure` и использование стратегии выполнения (выделено жирным шрифтом) с транзакцией. Код транзакции написан таким образом, что при выполнении повторной попытки вся транзакция запускается заново с самого начала.

Листинг 11.26 Запись транзакций при настройке стратегии выполнения

```
var connection = @"Server=(localdb)\mssqllocaldb;Database=… etc.";
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();

optionsBuilder.UseSqlServer(connection,
    option => option.EnableRetryOnFailure());
var options = optionsBuilder.Options;
using (var context = new Chapter09DbContext(options))
{
    var strategy = context.Database
        .CreateExecutionStrategy();
    strategy.Execute(() =>
    {
        try
        {
            using (var transaction = context
                .Database.BeginTransaction())
            {
                context.Add(new MyEntity());
                context.SaveChanges();
                context.Add(new MyEntity());
                context.SaveChanges();
                transaction.Commit();
            }
        }
        catch (Exception e)
        {
            //Здесь должна идти обработка ошибок;
            throw;
        }
    });
}
```

Настраивает базу данных для использования стратегии выполнения SQL, поэтому вам придется обрабатывать транзакции по-другому

Создает экземпляр IExecutionStrategy, использующий стратегию выполнения, которую мы применили для настройки DbContext

Важно превратить весь код транзакции в делегат Action, который можно вызывать

Остальная часть настройки транзакции и выполнение кода те же

ПРЕДУПРЕЖДЕНИЕ Код из листинга 11.26 безопасен, когда нужно повторить попытку. Говоря *безопасный*, я имею в виду, что код будет работать правильно. Но в некоторых случаях, напри-

мер когда изменяются данные за пределами действия повторной попытки стратегии выполнения, повторная попытка может вызвать проблемы. Очевидный пример – переменная `int count = 0`, определенная вне области действия повтора, которая увеличивается внутри действия. В таком случае значение переменной `count` снова будет увеличено, если случится повторная попытка. Помните об этом при проектировании транзакций, если используете повторную попытку со стратегией выполнения.

11.8.2 Изменение или написание собственной стратегии исполнения

В некоторых случаях вам может потребоваться изменить стратегию выполнения для базы данных. Если для поставщика базы данных (например, SQL Server) есть существующая стратегия выполнения, то можно изменить некоторые параметры, например количество повторных попыток или набор ошибок SQL, при которых нужно выполнить повтор.

Если вы хотите написать собственную стратегию выполнения, необходимо реализовать класс, наследующий от интерфейса `IExecutionStrategy`. Рекомендую обратить внимание на внутренний класс EF Core, `SqlServerExecutionStrategy`, в качестве шаблона. Этот шаблон можно найти на странице http://mng.bz/A1DK.

После того как вы написали собственный класс стратегии выполнения, можно настроить его для базы данных, используя метод `ExecuteStrategy` с параметром, как показано жирным шрифтом в следующем листинге.

Листинг 11.27 Настройка собственной стратегии выполнения в DbContext

```
var connection = this.GetUniqueDatabaseConnectionString();
var optionsBuilder =
    new DbContextOptionsBuilder<Chapter09DbContext>();

optionsBuilder.UseSqlServer(connection,
    options => options.ExecutionStrategy(
        p => new MyExecutionStrategy()));

using (var context = new Chapter09DbContext(optionsBuilder.Options))
{
  … и т.д.
```

Резюме

- Можно использовать свойство `State` сущности и флаг `IsModified` для отдельных свойств сущности, чтобы определить, что произойдет с данными при вызове метода `SaveChanges`.

- Можно повлиять на свойство `State` сущности и ее связи несколькими способами. Вы можете использовать методы DbContext `Add`, `Remove`, `Update`, `Attach` и `TrackGraph`, установить свойство `State` напрямую и отслеживать модификации.
- Свойство DbContext `ChangeTracker` предоставляет несколько способов обнаружения значения свойства `State` всех изменившихся сущностей. Эти методы полезны для маркировки сущностей датой создания, или последнего обновления сущности, или регистрации каждого изменения этого свойства для любой из отслеживаемых сущностей.
- В свойстве `Database` есть методы, позволяющие использовать строковые команды SQL для запросов к базе данных.
- Вы можете получить доступ к информации о сущностях и их связях, используя `Entry(entity).Metadata`, и о структуре базы данных, используя свойство `Model`.
- EF Core содержит систему, позволяющую предоставить возможность выполнения повторных попыток. Эта система может повысить надежность, выполняя повторные попытки запросов, если в вашей базе данных возникают ошибки подключения или другие случайные ошибки.

Для читателей, знакомых с EF6:

- EF Core изменила способы настройки свойства `State` сущности, основываясь на уроках, усвоенных в EF6.x. Теперь существует бóльшая вероятность установить для этого свойства правильное значение для действия, которое вы используете.
- EF Core представляет новый метод `TrackGraph`, который будет обходить граф связанных сущностей и вызывать ваш код, чтобы установить для свойства `State` каждой сущности нужное вам значение.
- То, как вы используете команды SQL в EF Core, отличается от того, как это делается в EF6.
- Свойства `Entry(entity).Metadata` и `Model` – это потрясающее улучшение в доступе к метаданным модели относительно EF6.x. Теперь вы можете получать доступ ко всем аспектам модели базы данных.
- Стратегия выполнения EF Core представляет собой улучшение стратегии выполнения EF6.x, поскольку EF Core может обрабатывать повторные попытки в транзакциях базы данных.

Часть III

Использование Entity Framework Core в реальных приложениях

В первой и второй частях мы подробнее познакомились с EF Core, и на каждом этапе я пытался привести примеры использования каждой функции или подхода. Теперь, в третьей части, мы создадим более сложную версию приложения Book App, а затем настроим его производительность. Будет здесь и новая информация. Например, в главе 16 мы поговорим о Cosmos DB, а в главе 17 – о модульном тестировании, но основное внимание в этой части уделяется использованию EF Core.

Я фрилансер. Мои клиенты хотят, чтобы их требования превратились в надежные, безопасные и высокопроизводительные приложения – и они нужны им быстро! Чтобы предоставить им такие приложения, я использую надежные, безопасные и высокопроизводительные подходы и библиотеки. Первые две главы этой части охватывают различные подходы, которые я усвоил за годы работы. Они позволяют мне быстро создавать приложения. Как сказал Кент Бек: «Делайте так, чтобы это работало, делайте это правильно, делайте это быстро».

Создав приложение в главах 12 и 13, мы перейдем к настройке производительности. Первоначальный вариант приложения Book App содержит около 700 настоящих книг, но в целях тестирования производительности мы создадим их копии, и у нас будет 100 000 книг

и более. Такое количество позволяет выявить проблемы, связанные с производительностью базы данных, и в двух с половиной главах мы улучшим ее, используя ряд методов.

Глава 16 посвящена применению Cosmos DB, чтобы окончательно настроить производительность Book App. В этой главе раскрываются различия между реляционными и нереляционными базами данных, чтобы вы лучше понимали, где и как их использовать.

Наконец, в главе 17 рассматривается модульное тестирование с упором на EF Core. Если задействована база данных, то модульное тестирование требует тщательного обдумывания, особенно если вы не хотите, чтобы модульный тест был медленным. Я поделюсь несколькими техниками и подходами, предложив созданный мной nuget пакет EfCore.TestSupport. Эта библиотека содержит функции настройки, которые помогут вам безопасно и быстро выполнить модульное тестирование приложений EF Core.

12 Использование событий сущности для решения проблем бизнес-логики

В этой главе рассматриваются следующие темы:

- типы событий, которые хорошо работают с EF Core;
- использование событий предметной области для запуска дополнительных бизнес-правил;
- использование событий интеграции для синхронизации двух частей приложения;
- реализация диспетчера событий и последующее его улучшение.

В программном обеспечении термин «событие» охватывает широкий спектр архитектур и паттернов. В целом он означает, что «Действие А запускает действие В». Вы уже видели события в главе 11, например события, в которых изменяется состояние сущности (раздел 11.4.4). Но эта глава о другом, совершенно ином типе события, которое я называю *событием сущности*, потому что оно находится в классах сущностей. Использование события сущности похоже на размещение сообщения в классе сущности, чтобы кто-нибудь позже прочитал его.

Цель событий сущности – запустить бизнес-логику, когда что-то меняется в классе сущности. В разделе 12.1.1 я привожу пример, в котором изменение сведений об адресе приводит к обновлению ставки налога с продаж. Этот пример реализуется путем обнаружения изменения в сведениях об адресе и отправки события сущности (сообще-

ния), которое инициирует некую бизнес-логику, обновляющую ставку налога с продаж для этого адреса.

Помимо событий сущности, нужны части, которые заставят их работать. В сердце подхода, использующего эти события, лежит код, который я называю *диспетчером событий*. Его задача – читать все события сущности и запускать конкретный бизнес-код (называемый *обработчиками событий*), связанный с каждым событием сущности. Каждый обработчик событий содержит конкретную бизнес-логику для этого события, а каждое сообщение события сущности предоставляет данные, необходимые обработчику.

Диспетчер событий запускается перед вызовом методов `SaveChanges` и `SaveChangesAsync`. Лучше всего переопределить эти методы, а затем запустить в них диспетчер событий. Я называю эти методы событийно-расширенными.

12.1 Использование событий для решения проблем бизнес-логики

Я придумал название «события сущности», но гораздо более умные люди придумали термины *события предметной области* и *события интеграции*, чтобы определить два варианта их использования. В этой главе вы узнаете о событиях предметной области и событиях интеграции, а также о ситуациях, в которых их можно использовать. После этого мы реализуем событийно-расширенные методы `SaveChanges` и `SaveChangesAsync`, которые можно использовать в приложениях.

12.1.1 Пример использования событий предметной области

Один из моих клиентов познакомил меня с событиями предметной области. Он использовал систему событий, которая обсуждалась в статье Джимми Богарда (http://mng.bz/oGNp). В ней описано, как добавлять события предметной области в EF Core. Я читал эту статью несколько лет назад и не понял ее, а вот мой клиент понял и с успехом использовал события предметной области. Видя, как события сущности используются в реальном приложении, я убедился в их полезности и стал использовать события предметной области для решения бизнес-требований и проблем, связанных с производительностью, в клиентском приложении. Следующий пример взят из одного из этих бизнес-требований.

Компания моего клиента занимается строительством домов на заказ в Соединенных Штатах, и каждый проект начинается с расчета стоимости работ, который нужно отправить заказчику. Строительство может идти где угодно, и штат, в котором выполняется работа, определяет налог с продаж. В результате налог с продаж приходилось пересчитывать, если происходило что-либо из нижеперечисленного:

- *появлялся новый расчет стоимости работ*. По умолчанию для нового расчета нет местоположения, поэтому бизнес-правило заключалось в том, чтобы назначить самый высокий налог с продаж до тех пор, пока не будет указано местоположение;
- *место работы было определено или изменено*. Налог с продаж приходилось пересчитывать, а работа отдела продаж заключалась в выборе местоположения из списка известных местоположений;
- *менялся адрес местоположения*. Все расчеты, связанные с этим местоположением, приходилось вести заново, чтобы убедиться, что налог с продаж был правильным.

Можно было добавить бизнес-логику для всех этих действий, но из-за этого клиентская часть стала бы более сложной, и легко было бы пропустить область, где местоположение менялось. Тогда налог с продаж был бы неправильным. Решение состояло в том, чтобы использовать события, которые возникали, если местоположение в расчете стоимости добавлялось или обновлялось, и это прекрасно работало. Изменение в классе сущности `Location` создавало событие предметной области, запускавшее обработчик события, который пересчитывал налог с продаж в расчете стоимости. Каждому событию предметной области требовалась немного разная часть бизнес-логики плюс общий сервис по расчету налога. На рис. 12.1 показан пример того, что может произойти, если адрес местоположения изменится.

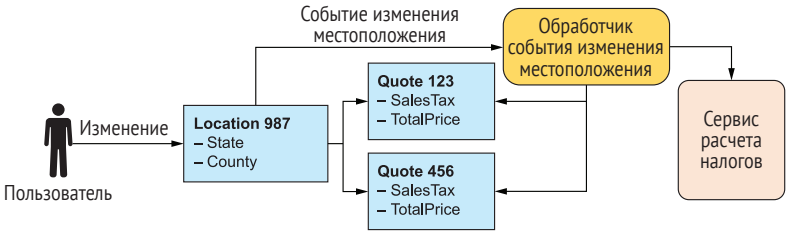


Рис. 12.1 Вместо того чтобы добавлять код в клиентской части для выполнения бизнес-логики, когда местоположение меняется, можно перехватить изменение в классе сущности и добавить в класс событие предметной области. При вызове метода `SaveChanges` добавленный в него фрагмент кода просматривает все события предметной области и запускает соответствующий обработчик событий, чтобы убедиться, что для всех открытых расчетов стоимости работ налог с продажи пересчитан

Сейчас я не буду вдаваться в подробности того, как работает этот пример, поскольку в этом разделе описано, где и почему события полезны. Достаточно сказать, что в разделе 12.4 мы напишем код для обработки события сущности и будем улучшать его по мере изучения этого подхода.

12.1.2 Пример событий интеграции

Второй вариант использования события сущности – более сложная ситуация. В главе 13 вы узнаете о способах повышения производительности доступа к базе данных в EF Core. Один из этих подходов – предварительно вычислить данные, которые необходимо показать пользователю, и сохранить их в другой базе, используемой только для отображения данных пользователю. Этот подход улучшает скорость чтения и масштабируемость.

Например, обычные SQL-запросы для приложения Book App вычисляют среднюю оценку книги путем динамического расчета среднего значения по всем отзывам (`Reviews`). Такой метод отлично подходит для небольшого количества книг и отзывов, но при большом количестве сортировка по средним оценкам может стать медленной. В главе 16 мы будем использовать паттерн базы данных Command and Query Responsibility Segregation (CQRS) для хранения предварительно вычисленных данных в отдельной базе данных, доступной для чтения. Проблема состоит в том, чтобы убедиться, что SQL (база данных для операций записи) и Cosmos DB (база данных для операций чтения) всегда «идут в ногу».

Я использую это решение в главе 16: при выполнении записи в базу данных SQL запускается транзакция, содержащая обновление для базы данных SQL и обновление для Cosmos DB. Если одна из баз данных выйдет из строя, запись не будет выполнена. А это означает, что согласованность данных не будет нарушена. На рис. 12.2 показано, как может работать это решение.

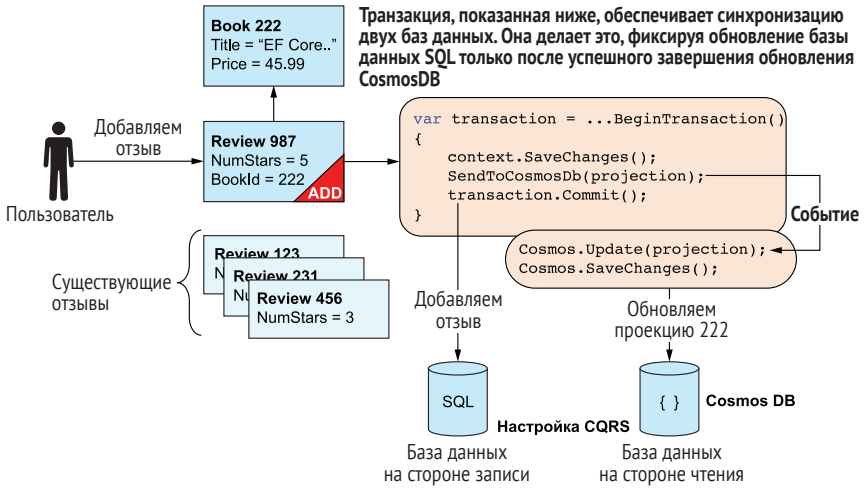


Рис. 12.2 База данных с CQRS, использующая реляционную базу данных в качестве основного хранилища и Cosmos DB как базу данных для чтения для повышения производительности. Вопрос в том, как удостовериться, что у этих баз нет расхождений, – в данном случае проекция Book в Cosmos DB соответствует тому, что есть в базе данных SQL. На этом рисунке показан пример использования событий интеграции для совместной работы кода, обрабатывающего реляционную базу данных, и кода, обрабатывающего базу данных Cosmos DB

12.2 Определяем, где могут быть полезны события предметной области и интеграции

Два примера, которые вы видели, используют события по-разному; пример с налогом с продаж сосредоточен в определенной части кода, относящейся к клиентам и расчетам стоимости, а пример с CQRS связывает две совершенно разные части приложения. Этим двум типам событий были даны имена, преимущественно DDD-сообществом, но вы увидите, что события также можно использовать и в обычных типах сущностей, не относящихся к предметно-ориентированному проектированию (domain driven design, DDD).

> **ПРИМЕЧАНИЕ** Я расскажу, как применить предметно-ориентированное проектирование к классам сущностей EF Core, в главе 13, а в этой главе вы узнаете, как использовать события в типах сущностей, не относящихся к предметно-ориентированному проектированию.

В предметно-ориентированном проектировании много говорится об ограниченном контексте (bounded context), который представляет определенную часть программного обеспечения, где конкретные термины, определения и правила применяются согласованным образом. Ограниченный контекст касается применения принципа разделения ответственностей на макроуровне. Например, приложение Book App из третьей части разбито на несколько ограниченных контекстов: один обрабатывает отображение книг, используя реляционную базу данных, другой предоставляет способ отображения книг с помощью нереляционной базы данных, а третий обрабатывает заказ пользователя. Таким образом, используя термин «ограниченный контекст», можно классифицировать два типа событий следующим образом:

- пример с налогом с продаж называется событием предметной области, потому что он работает исключительно в рамках одного ограниченного контекста;
- пример с CQRS называется событием интеграции, потому что переходит из одного ограниченного контекста в другой.

> **ПРИМЕЧАНИЕ** Ограниченные контексты подробнее рассматриваются в главе 13.

12.3 Где можно использовать события с EF Core?

Я не предлагаю вам делать все подряд, используя события сущности, но думаю, что эти события – хороший подход к обучению. Где бы можно было их использовать? Лучше всего на это ответят примеры:

- задание или изменение адреса (`Address`) инициирует перерасчет налога с продаж для `Quote`;
- создание заказа (`Order`) инициирует проверку необходимости пополнения склада (`Stock`);
- обновление `Book` инициирует обновление проекции (`Projection`) этой книги в другой базе данных;
- получение платежа (`Payment`) для оплаты задолженности инициирует закрытие учетной записи (`Account`);
- отправка сообщения (`Message`) во внешний сервис (`external service`).

В каждом примере есть две сущности в скобках. Эти сущности разные и не являются тесно связанными: `Address/Quote`, `Order/Stock`, `Book/Projection`, `Payment/Account` и `Message/external service`. Когда я говорю, что классы не являются тесно связанными, то имею в виду, что второй класс не зависит от первого. Если, например, запись `Address` удалить, то запись `Quote` останется.

ПРИМЕЧАНИЕ Хороший показатель того, что события предметной области могут вам помочь, – ситуация, когда ваша бизнес-логика будет работать с двумя разными группами данных.

Во всех этих случаях с первым классом можно было бы обращаться стандартным образом (т. е. не использовать события сущности), а событие предметной области могло бы запускать обработчик событий, чтобы обновить второй класс. И наоборот, события бесполезны, когда классы сущностей уже тесно связаны. Например, вы не будете использовать события для настройки каждой позиции `LineItem` в заказе `Order`, потому что эти два класса тесно связаны друг с другом.

Еще один случай, когда события могут оказаться полезны, – когда вы хотите добавить новую функциональность в некий существующий код, но не хотите изменять существующие методы и бизнес-логику. Если это изменение не меняет существующий код, то можно использовать события, даже если два класса сущности тесно связаны. В главе 15 есть хороший пример, улучшающий производительность существующего приложения Book App. Я не хочу менять существующий код, который работает, но хочу добавить кешированные значения в класс сущности `Book`, и использование событий предметной области – отличное решение.

Прочитав все это, вполне вероятно, что вы не будете использовать много событий предметной области. Например, в системе, из которой взят пример с налогом с продаж, было всего 20 событий предметной области, но некоторые из них были критически важны для функций и особенно для производительности приложения.

События интеграции встречаются еще реже; они полезны только тогда, когда у вас есть два ограниченных контекста, которые должны работать совместно. Но если нужно синхронизировать две разные части приложения, то события интеграции – это один из лучших подходов, которые можно использовать.

В целом я считаю подобные события настолько полезными, что создал библиотеку `EfCore.GenericEventRunner`, чтобы с легкостью добавлять в приложение события (предметной области и интеграционные), когда они мне нужны. Но прежде чем я расскажу, как реализовать такую систему, рассмотрим плюсы и минусы использования этих событий.

12.3.1 Плюс: следует принципу разделения ответственностей

Уже описанные системы позволяют запускать отдельные бизнес-правила при изменении в классе сущности. В примере с изменением местоположения и налогом с продаж две сущности связаны неочевидным образом; изменение местоположения проведения работ вызывает перерасчет налога с продаж для всех связанных расчетов. Когда вы применяете принцип разделения ответственностей, эти два бизнес-правила следует разделять.

Можно было бы создать некую бизнес-логику для обработки обоих бизнес-правил, но это усложнит простое обновление свойств в адресе. Инициируя событие при изменении свойств `State/County`, можно сохранить простоту обновления адреса, позволяя событию обрабатывать вторую часть.

12.3.2 Плюс: делает обновления базы данных надежными

Разработка кода, обрабатывающего события предметной области, такова, что исходное изменение, инициирующее событие, и изменения, применяемые к классам сущностей через вызываемый обработчик событий, сохраняются в одной и той же транзакции. На рис. 12.3 показан этот код в действии.

Как вы увидите в разделе 12.5, реализация события интеграции также является надежной. Если событие интеграции завершится неудачей, обновление базы данных будет отменено. Это гарантирует, что локальная база данных, внешний сервис и другая база данных работают синхронно.

12.3.3 Минус: делает приложение более сложным

Один из недостатков использования событий состоит в том, что ваш код будет более сложным. Даже если вы используете такую библиотеку, как `EfCore.GenericEventRunner`, для управления событиями вам все равно придется создавать свои события, добавлять их в классы сущностей и писать свои обработчики событий, что требует больше кода, нежели создание сервисов для бизнес-логики, как описано в главе 4.

Но компромисс при работе с событиями, требующими бóльшего количества кода, заключается в том, что две части бизнес-логики не являются связанными. Например, изменения адреса превращаются в простое обновление, в то время как событие гарантирует, что код для перерасчета налогов будет вызван. Это снижает сложность бизнес-логики, с которой приходится иметь дело разработчику.

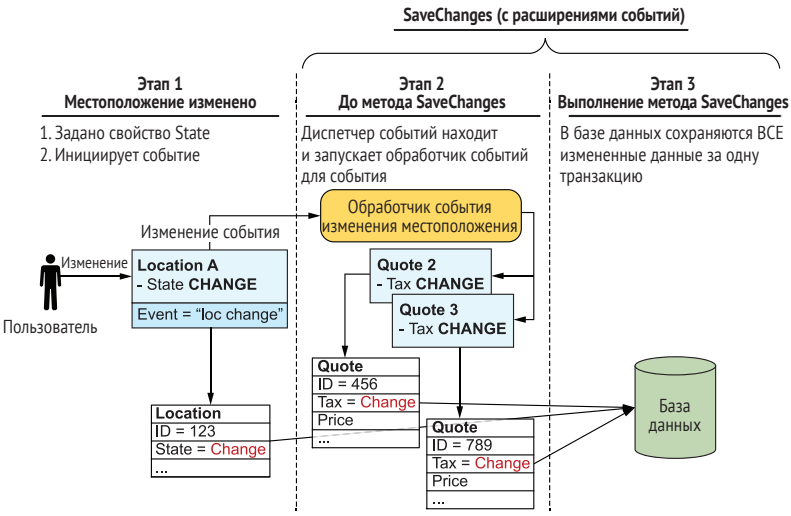


Рис. 12.3 Система событий предметной области сохраняет первоначальное обновление `Location`, инициировавшее событие, и изменения, внесенные в классы сущностей `Quote`, в одной транзакции. База данных сохранит все изменения за один раз, поэтому обновления всегда будут согласованы

12.3.4 Минус: усложняет отслеживание потока исполнения кода

Бывает непросто понять код, который писали не вы или писали вы, но какое-то время назад. Есть одна полезная функция VS/VS Code, которую я использую, – это Go to Implementation (переход к реализации). Она позволяет мне переходить к коду метода, чтобы я мог покопаться в коде и понять, как работает каждая часть, прежде чем я ее поменяю.

Вы можете делать то же самое, когда используете события, но эта техника добавляет еще один дополнительный шаг, прежде чем вы перейдете к коду. В случае с примером изменения налога с продаж на рис. 12.1 нужно будет щелкнуть на класс `LocationChangedEvent`, чтобы найти `LocationChangedEventHandler` с бизнес-кодом, который вы ищете, – всего лишь дополнительный шаг, но его бы делать не понадобилось, если бы вы не использовали события.

12.4 Реализация системы событий предметной области с EF Core

В этом разделе мы реализуем систему событий предметной области в EF Core. Сперва мы добавим возможность хранить события сущности в классах сущностей. После этого переопределим метод DbContext `SaveChanges`, чтобы у нас была дополнительная логика для извлечения событий сущности, поиска и запуска соответствующего обработчика событий.

На рис. 12.4 показаны код и этапы, необходимые для реализации этой системы. Здесь используется пример, показанный на рис. 12.1, где свойство `State` сущности `Location` меняется. В этом примере два объекта `Quote` связаны с этим местоположением, поэтому их свойство `SalesTax` нужно обновить, дабы получить правильный налог с продаж для этого местоположения.

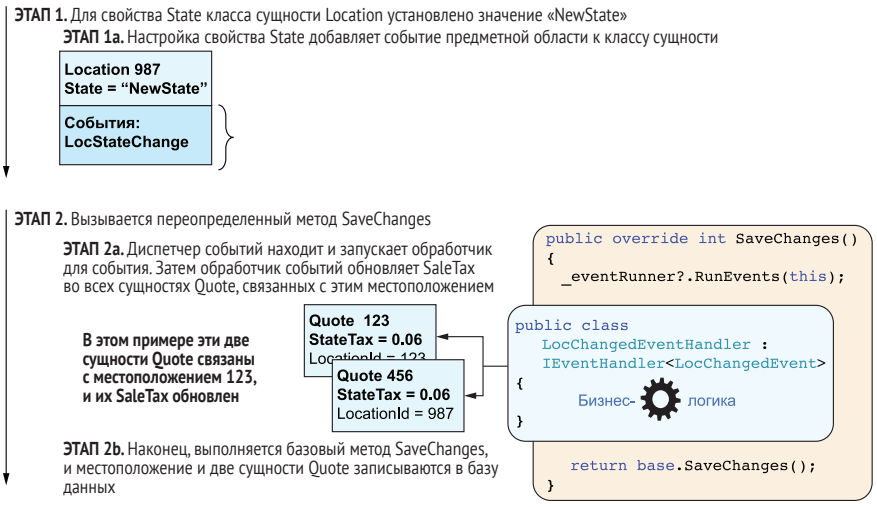


Рис. 12.4 Этап 1 показывает, что изменение свойства `State` сущности `Location` приводит к тому, что событие предметной области добавляется в класс сущности `Location`. На этапе 2, когда вызывается переопределенный метод `SaveChanges`, будут читаться любые события предметной области в отслеживаемых сущностях, а затем будет находиться и запускаться соответствующий обработчик для каждого события предметной области. В этом примере обработчик событий обновляет свойство `SalesTax` во всех `Quote`, связанных с этим местоположением

Чтобы реализовать такую систему событий предметной области, добавьте в свое приложение следующий код:

1. Создайте несколько классов событий предметной области, которые будут вызваны.
2. Добавьте код в классы сущностей, где будут храниться события предметной области.

3. Измените код в классе сущности, чтобы обнаружить изменение, при котором вы хотите вызвать событие.
4. Создайте несколько обработчиков, соответствующих событиям. Эти обработчики могут изменить вызывающий класс сущности или получить доступ к базе данных или бизнес-логике, чтобы выполнить бизнес-правила, для обработки которых они предназначены.
5. Создайте диспетчер событий, который находит и запускает правильный обработчик, соответствующий каждому найденному событию.
6. Добавьте диспетчер событий в DbContext и переопределите метод `SaveChanges` (и `SaveChangesAsync`) в DbContext.
7. Когда диспетчер событий завершит работу, выполните базовый метод `SaveChanges`, который обновит базу данных исходными и любыми последующими изменениями, примененными обработчиками событий.
8. Зарегистрируйте диспетчер событий и все обработчики событий.

Далее мы выполним эти шаги, чтобы реализовать все этапы этого подхода.

12.4.1 Создайте несколько классов событий предметной области, которые нужно будет вызвать

Создание события состоит из двух частей. Во-первых, у него должен быть интерфейс, позволяющий диспетчеру событий ссылаться на него. Этот интерфейс может быть пустым, представляя событие. (В этом примере я называю данный интерфейс `IDomainEvent`.) Я использую его для представления события предметной области внутри диспетчера событий. Каждое событие приложения содержит данные, которые относятся только к бизнес-требованиям. В следующем листинге показан класс `LocationChangedEvent`, которому требуется только класс сущности `Location`.

Листинг 12.1 Класс `LocationChangedEvent` с данными, которые нужны обработчику событий

```
public class LocationChangedEvent : IDomainEvent
{
    public LocationChangedEvent(Location location)
    {
        Location = location;
    }

    public Location Location { get; }
}
```

Класс события должен наследовать интерфейс IDomainEvent. Диспетчер событий использует этот интерфейс как маркер события предметной области

Обработчику событий требуется Location, чтобы обновлять Quote

Каждое событие должно отправлять данные, необходимые обработчику для выполнения. После этого задача обработчика событий – запустить некую бизнес-логику, используя данные, предоставленные событием.

12.4.2 Добавьте код в классы сущностей, где будут храниться события предметной области

Класс сущности должен содержать список событий. Эти события не записываются в базу данных, но существуют для того, чтобы диспетчер событий смог прочитать их через метод. В следующем листинге показан класс, от которого сущность может наследовать, чтобы добавить функцию событий.

Листинг 12.2 Класс, от которого наследуют классы сущностей, чтобы создать события

```
public class AddEventsToEntity : IEntityEvents
{
    private readonly List<IDomainEvent>
        _domainEvents = new List<IDomainEvent>();

    public void AddEvent(IDomainEvent domainEvent)
    {
        _domainEvents.Add(domainEvent);
    }

    public ICollection<IDomainEvent>
        GetEventsThenClear()
    {
        var eventsCopy = _domainEvents.ToList();
        _domainEvents.Clear();
        return eventsCopy;
    }
}
```

Интерфейс IEntityEvents определяет метод GetEventsThenClear для диспетчера событий

Список событий IDomainEvent сохраняется в поле

Метод AddEvent используется для добавления новых событий в список _domainEvents

Этот метод вызывается диспетчером событий, чтобы получить события и затем очистить список

Класс сущности может вызвать метод `AddEvent`, а диспетчер событий может получить события предметной области через метод `GetEventsThenClear`. Кроме того, получение событий предметной области удаляет события в классе сущности, потому что эти сообщения приведут к выполнению обработчика, а нам нужно, чтобы обработчик запускался только один раз для каждого события предметной области. Помните, что эти события не похожи на события C#; события предметной области – это сообщения, передаваемые диспетчером событий через классы сущностей, а нам нужно, чтобы сообщение использовалось только один раз.

12.4.3 Измените класс сущности, чтобы обнаружить изменение, при котором вызывается событие

Обычно событие – это когда что-то изменяется или достигает определенного уровня. EF Core позволяет использовать резервные поля, которые упрощают перехват изменений в скалярных свойствах. В следующем листинге показан класс сущности `Location`, создающий событие предметной области при изменении свойства `State`.

Листинг 12.3 Класс сущности `Location` создает событие предметной области, если свойство `State` изменяется

```
public class Location : AddEventsToEntity
{
    public int LocationId { get; set; }
    public string Name { get; set; }

    private string _state;

    public string State
    {
        get => _state;
        set
        {
            if (value != _state)
                AddEvent(
                    new LocationChangedEvent(this));
            _state = value;
        }
    }
}
```

Этот класс сущности наследует от класса AddEventsToEntity, чтобы получить возможность использовать события

Эти обычные свойства не генерируют события при изменении

Резервное поле содержит реальное значение данных

Метод записи вызовет событие LocationChangedEvent, если значение свойства State изменится

Этот код добавит событие LocationChangedEvent в класс сущности, если значение свойства State изменится

ПРИМЕЧАНИЕ Навигационные свойства коллекции немного сложнее проверить на предмет наличия изменений, но классы сущностей, разработанные в стиле предметно-ориентированного проектирования (описанные в главе 13), упрощают эту проверку.

12.4.4 Создайте обработчики событий, соответствующие событиям предметной области

Обработчики событий являются ключом к использованию событий в приложении. Каждый обработчик содержит некую бизнес-логику, которую необходимо выполнить при обнаружении конкретного события. Чтобы диспетчер событий работал, у каждого обработчика должна быть одна и та же сигнатура, определяемая интерфейсом

`IEventHandler<T> where T : IDomainEvent`, который я создал для этого примера. В следующем листинге показан обработчик, который обновляет `SalesTax` в каждом `Quote`, связанном с измененным местоположением.

Листинг 12.4 Обработчик событий обновляет налог с продаж для экземпляров Quote, связанных с этим местоположением

```
public class LocationChangedEventHandler
    : IEventHandler<LocationChangedEvent>
    {
    private readonly DomainEventsDbContext _context;
    private readonly
        ICalcSalesTaxService _taxLookupService;

    public LocationChangedEventHandler(
        DomainEventsDbContext context,
        ICalcSalesTaxService taxLookupService)
    {
        _context = context;
        _taxLookupService = taxLookupService;
    }

    public void HandleEvent
        (LocationChangedEvent domainEvent)
    {
        var salesTaxPercent = _taxLookupService
            .GetSalesTax(domainEvent.Location.State);

        foreach (var quote in _context.Quotes.Where(
            x => x.WhereInstall == domainEvent.Location))
        {
            quote.SalesTaxPercent = salesTaxPercent;
        }
    }
}
```

Этот класс должен быть зарегистрирован как сервис с помощью внедрения зависимостей

У каждого обработчика должен быть интерфейс IEventHandler<T>, где T – тип класса события

Для этого конкретного обработчика необходимо два класса, зарегистрированных с помощью внедрения зависимостей

Диспетчер событий будет использовать внедрение зависимостей для получения экземпляра этого класса и заполнит параметры конструктора

Метод из IEventHandler<T>, который диспетчер событий будет выполнять

Использует еще один сервис для правильного расчета налога с продаж

Устанавливает SalesTax для каждого Quote, связанного с этим местоположением

Ключевой момент: обработчик событий зарегистрирован как сервис, поэтому диспетчер событий может получить экземпляр класса обработчика, используя внедрение зависимостей. У класса обработчика событий такой же доступ к сервисам внедрения зависимостей, что и у обычной бизнес-логики. В данном случае `LocationChangedEventHandler` внедряет DbContext приложения и сервис `ICalcSalesTaxService`.

12.4.5 Создайте диспетчер событий, который находит и запускает правильный обработчик событий

Диспетчер событий – это сердце системы событий: его задача – сопоставить каждое событие с обработчиком событий, а затем вызвать метод обработчика событий, предоставив событие в качестве параметра. Этот процесс использует `ServiceProvider` от NET Core для получения экземпляра обработчика событий, что позволяет обработчикам событий получать доступ к другим сервисам. На рис. 12.5 изображено визуальное представление того, что делает диспетчер событий.

ПРИМЕЧАНИЕ Если в приложении нет функции внедрения зависимостей, то можно заменить ее, вручную написав оператор `switch` с кодом для создания каждого менеджера событий. С такой техникой сложнее справиться, но она сработает.

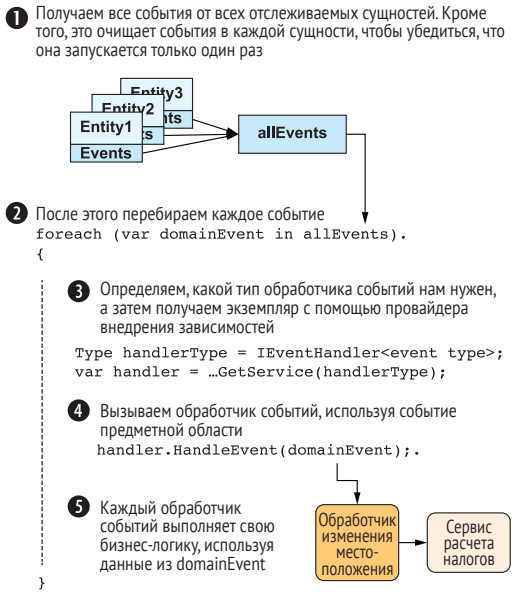


Рис. 12.5 Диспетчер событий собирает события из каждой отслеживаемой сущности, у которой есть интерфейс `IEntityEvents`; после этого для каждого события он получает экземпляр соответствующего обработчика и вызывает обработчик с событием в качестве параметра. Наконец, каждый обработчик событий запускает свою бизнес-логику, используя данные, полученные в событии

В следующем листинге показан код диспетчера событий. Он довольно сложен, потому что его проектирование требует использования обобщенных типов.

Листинг 12.5 Диспетчер событий, который вызывается из переопределенного метода SaveChanges

```
public class EventRunner : IEventRunner
{
    private readonly IServiceProvider _serviceProvider;

    public EventRunner(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public void RunEvents(DbContext context)
    {
        var allEvents = context.
            ChangeTracker.Entries<IEntityEvents>()
            .SelectMany(x => x.Entity.GetEventsThenClear());

        foreach (var domainEvent in allEvents)
        {
            var domainEventType = domainEvent.GetType();
            var eventHandleType = typeof(IEventHandler<>)
                .MakeGenericType(domainEventType);
            var eventHandler =
                _serviceProvider.GetService(eventHandleType);
            if (eventHandler == null)
                throw new InvalidOperationException(
                    $"Could not find an event handler")

            var handlerRunnerType = typeof(EventHandlerRunner<>)
                .MakeGenericType(domainEventType);
            var handlerRunner = ((EventHandlerRunner)
                Activator.CreateInstance(
                    handlerRunnerType, eventHandler));

            handlerRunner.HandleEvent(domainEvent);
        }
    }
}
```

Диспетчеру событий нужен интерфейс, чтобы его можно было зарегистрировать с помощью внедрения зависимостей

Диспетчеру событий нужен ServiceProvider для получения экземпляра обработчиков событий

Считывает все события и очищает события сущности, чтобы предотвратить повторную обработку событий

Перебирает каждое найденное событие

Получает тип интерфейса соответствующего обработчика событий

Использует EventHandlerRunner для запуска обработчика событий

Использует поставщика внедрения зависимостей для создания экземпляра обработчика событий и возвращает ошибку, если он не найден

Создает EventHandlerRunner, необходимый для запуска обработчика событий

В следующем листинге показаны классы `EventHandlerRunner` и `EventHandlerRunner<T>`. Они нужны, потому что определение обработчика событий обобщенное и нельзя вызвать его напрямую. Можно обойти эту проблему, создав класс, принимающий в своем конструкторе обобщенный обработчик событий, у которого есть необобщенный метод (абстрактный класс `EventHandlerRunner`), который можно вызвать.

Листинг 12.6 Класс EventHandlerRunner, запускающий обобщенный обработчик событий

```
internal abstract class EventHandlerRunner
{
    public abstract void HandleEvent
        (IDomainEvent domainEvent);
}

internal class EventHandlerRunner<T> : EventHandlerRunner
    where T : IDomainEvent
{
    private readonly IEventHandler<T> _handler;

    public EventHandlerRunner(IEventHandler<T> handler)
    {
        _handler = handler;
    }

    public override void HandleEvent
        (IDomainEvent domainEvent)
    {
        _handler.HandleEvent((T)domainEvent);
    }
}
```

Определив необобщенный метод, можно запустить обобщенный обработчик событий

Использует EventHandlerRunner<T> для определения типа EventHandlerRunner

Класс EventHandlerRunner создается с экземпляром обработчика событий для запуска

Метод, переопределяющий метод HandleEvent абстрактного класса

12.4.6 Переопределите метод SaveChanges и вставьте вызов диспетчера событий перед вызовом этого метода

После этого мы переопределяем методы `SaveChanges` и `SaveChangesAsync`, чтобы код диспетчера событий выполнялся перед выполнением базовой реализации этих методов. Любые изменения, вносимые обработчиками событий в сущности, сохраняются с исходными изменениями, вызвавшими события. Этот момент действительно важен: изменения, внесенные в сущности кодом, не относящимся к событиям, сохраняются со всеми изменениями, внесенными обработчиками событий. Если при сохранении данных в базе возникает проблема (например, было выброшено исключение параллелизма), то ни одно из изменений не будет записано в базу данных, поэтому оба типа изменений сущности – где используется код, не относящийся к событиям, и код обработчика событий – не приведут к нарушению синхронизации при использовании CQRS. В следующем листинге показано, как внедрить диспетчер событий через конструктор DbContext приложения, а затем использовать его внутри переопределенного метода `SaveChanges`.

Листинг 12.7 DbContext приложения с переопределенным методом SaveChanges

```
public class DomainEventsDbContext : DbContext
{
    private readonly IEventRunner _eventRunner;

    public DomainEventsDbContext(
        DbContextOptions<DomainEventsDbContext> options,
        IEventRunner eventRunner = null)
        : base(options)
    {
        _eventRunner = eventRunner;
    }

    //… Свойство DbSet<T> не указано;

    public override int SaveChanges
        (bool acceptAllChangesOnSuccess)
    {
        _eventRunner?.RunEvents(this);
        return base.SaveChanges(acceptAllChangesOnSuccess);
    }

    //…  Переопределенный метод SaveChangesAsync не указан;
}
```

Содержит диспетчер событий, который внедряется через конструктор класса

У конструктора теперь есть второй параметр, который средство внедрения зависимостей заполняет экземпляром диспетчера событий

Вы переопределяете метод SaveChanges, чтобы можно было запустить диспетчер событий до «настоящего» метода SaveChanges

Запускает диспетчер событий

Запускает base.SaveChanges

ПРИМЕЧАНИЕ Есть две версии методов `SaveChanges` и `SaveChangesAsync`, но вам необходимо переопределить только одну из них. Например, нужно переопределить только `int SaveChanges (bool acceptAllChangesOnSuccess)`, потому что метод `SaveChanges` без параметров вызывает метод `SaveChanges`, где для параметра `acceptAllChangesOnSuccess` задано значение `true`.

12.4.7 Зарегистрируйте диспетчер событий и все обработчики событий

Последняя часть – это регистрация диспетчера событий и обработчиков событий в средстве внедрения зависимостей. Диспетчер событий полагается на внедрение зависимостей, чтобы предоставить экземпляры обработчиков событий, используя их интерфейсы. Кроме того, DbContext приложения нужно, чтобы диспетчер событий был внедрен в параметр конструктора типа `IEventRunner`. Когда диспетчер и обработчики событий регистрируются наряду со всеми сервисами, необходимыми обработчикам событий (например, сервис калькулятора налога с продаж), диспетчер событий будет работать. В этом простом примере можно зарегистрировать несколько классов и интерфейсов вручную с помощью средства внедрения зависимостей NET Core, как показано в следующем листинге.

Листинг 12.8 Регистрация диспетчера событий и обработчиков событий вручную в ASP.NET Core

```
public void ConfigureServices(IServiceCollection services)
{
    //… Остальные регистрации не указаны;

    services.AddTransient<IEventRunner, EventRunner>();

    services.AddTransient<IEventHandler<LocationChangedEvent>,
            LocationChangedEventHandler>();
    services.AddTransient<IEventHandler<QuoteLocationChangedEvent>,
        QuoteLocationChangedEventHandler>();

    services.AddTransient<ICalcSalesTaxService,
        CalcSalesTaxService>();
}
```

Вы регистрируете интерфейсы/классы с помощью поставщика внедрения зависимостей NET – в данном случае в приложении ASP.NET Core

Регистрирует диспетчер событий, который будет внедрен в DbContext приложения

Регистрирует все ваши обработчики событий

Необходимо зарегистрировать все сервисы, которые будут использовать ваши обработчики событий

Хотя ручная регистрация работает, есть более подходящий способ – автоматизировать поиск и регистрацию обработчиков событий. В листинге 12.9 показан метод расширения, который будет регистрировать диспетчер событий и все обработчики событий в каждой предоставленной вами сборке. В следующем фрагменте кода показано, как его вызвать:

```
services.RegisterEventRunnerAndHandlers(
    Assembly.GetAssembly(
        typeof(LocationChangedEventHandler)));
```

В следующем листинге показан код `RegisterEventRunnerAndHandlers`.

Листинг 12.9 Автоматическая регистрация диспетчера событий и обработчиков событий

```
public static void RegisterEventRunnerAndHandlers(
    this IServiceCollection services,
    params Assembly[] assembliesToScan)
{
    services.AddTransient<IEventRunner, EventRunner>();
    foreach (var assembly in assembliesToScan)
    {
        services.RegisterEventHandlers(assembly);
    }
}
```

Для этого метода требуется коллекция сервисов NET Core для регистрации зависимостей

Предоставляем одну или несколько сборок для сканирования

Регистрирует диспетчер событий

Вызывает метод для поиска и регистрации обработчика событий в сборке. Находит и регистрирует все классы, которые наследуют интерфейс IEventHandler<T>

```
private static void RegisterEventHandlers(
    this IServiceCollection services,
    Assembly assembly)
{
    var allGenericClasses = assembly.GetExportedTypes()
        .Where(y => y.IsClass && !y.IsAbstract
           && !y.IsGenericType && !y.IsNested);
    var classesWithIHandle =
        from classType in allGenericClasses
        let interfaceType = classType.GetInterfaces()
            .SingleOrDefault(y =>
                y.IsGenericType &&
                y.GetGenericTypeDefinition() ==
                    typeof(IEventHandler<>))
        where interfaceType != null
        select (interfaceType, classType);

    foreach (var tuple in classesWithIHandle)
    {
        services.AddTransient(
            tuple.interfaceType, tuple.classType);
    }
}
```

Находит и регистрирует все классы с интерфейсом IEventHandler<T>

Находит все классы в сборке, которые могут быть обработчиком событий

Находит все классы с интерфейсом IEventHandler<T>, а также тип интерфейса

Регистрирует каждый класс с его интерфейсом

ПРИМЕЧАНИЕ Код `RegisterEventRunnerAndHandlers` не будет регистрировать сервисы `CalcSalesTaxService`, потому что он ищет только обработчики событий. Но класс `CalcSalesTaxService` – это обычный сервис, т. е. класс с необобщенным интерфейсом, как и любой другой сервис. В главе 5, и в особенности в разделе 5.7.3, показано, как зарегистрировать такие виды сервисов.

Вот и все! Мы добавили в свое приложение события предметной области и теперь готовы к работе. Будем использовать эти события в главе 15 как один из способов улучшить производительность запросов к базе данных за счет обновления значений кеша при добавлении или удалении отзывов. Кроме того, можно увидеть это в действии в приложении Book App, щелкнув по ссылке SQL (cached) в меню.

12.5 Внедрение системы событий интеграции с EF Core

Теперь, когда вы увидели, как работают события предметной области, перейдем к событиям интеграции. События интеграции проще реализовать, чем события предметной области, но сложнее спроектировать, потому что они работают в нескольких ограниченных контекстах (см. раздел 12.2).

Есть много способов реализовать их, но наша книга посвящена EF Core, поэтому в этом разделе основное внимание уделяется использованию событий интеграции в транзакции базы данных в методе SaveChanges. Цель состоит в том, чтобы гарантировать, что база данных обновляется, только если событие интеграции было успешным.

Я привел один пример в разделе 12.2: объединение обновления базы данных SQL с соответствующим обновлением базы данных CQRS на стороне чтения. Этот пример работает, поскольку ядро пытается обновить базу данных CQRS на стороне чтения, только если обновление SQL было успешным, и фиксирует обновление SQL лишь в том случае, если база данных на стороне чтения CQRS была успешной. Таким образом, две базы данных будут содержать одни и те же данные. Можно обобщить этот пример, выделив две части, которые должны работать, чтобы действие было успешным:

- не отправлять событие интеграции, если обновление базы данных не сработало;
- не фиксировать обновление базы данных, если событие интеграции не сработало.

Теперь реализуем код, следующий подходу с событием интеграции. Простой пример: предположим, что вы создаете новый сервис, отправляющий клиентам их заказ на приобретение конструктора Lego курьером в тот же день. Вы не хотите разочаровывать своих клиентов, поэтому должны быть уверены, что у вас на складе есть товар и есть курьер, который может доставить заказ немедленно. Вся эта система изображена на рис. 12.6.

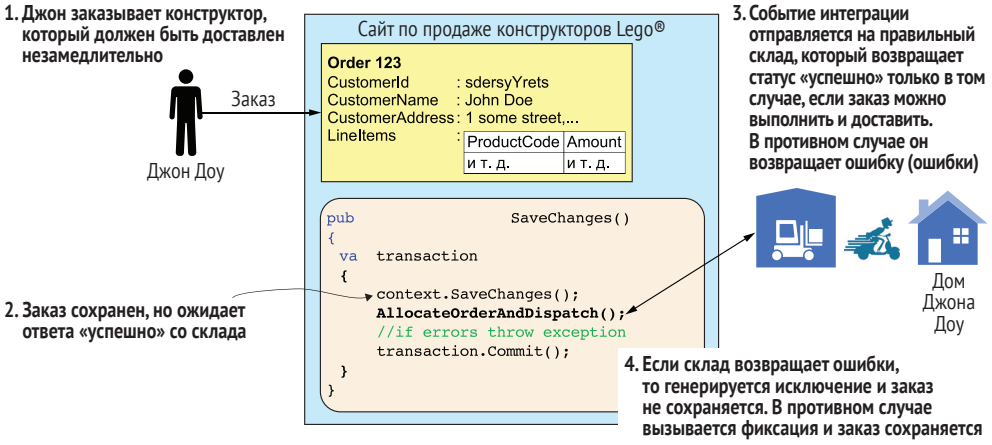


Рис. 12.6 Использование события интеграции, чтобы убедиться, что вы можете доставить заказ, прежде чем сохранить его. Чтобы реализовать это событие, вы переопределяете метод SaveChanges в DbContext приложения и обрабатываете все новые заказы, используя транзакцию. Заказ Order проверяется и сохраняется; после этого событие интеграции отправляет содержимое заказа на склад. Если на складе есть все необходимые товары и у вас есть курьер, чтобы доставить заказ клиенту, возвращается сообщение об успешном завершении, после чего Order фиксируется в базе данных. При возникновении ошибок заказ не записывается в базу данных, а клиент видит на экране ошибку

Есть два варианта обнаружить и обработать события интеграции в DbContext приложения:

- вы внедряете сервис непосредственно в DbContext приложения, который сам определяет, произошло ли конкретное событие, путем обнаружения свойства сущности `State`. Вторая часть вызывается только в том случае, если первый метод говорит, что ее нужно вызывать;
- можно использовать подход, аналогичный диспетчеру событий, который вы использовали для событий предметной области, но обработка другого типа событий будет запускаться в транзакции после вызова базового метода `SaveChanges`.

В большинстве случаев событий интеграции не так много, поэтому первый вариант быстрее; он обходит систему событий, которую вы добавили в сущность для событий предметной области, и выполняет собственное обнаружение события. Это простой подход, который хранит весь код в одном месте, но может стать громоздким, если вам нужно обнаружить и обработать несколько событий.

Второй вариант – это расширение диспетчера событий и событий предметной области, которое использует аналогичное создание события интеграции, когда в сущности что-то меняется. В этом конкретном случае код создаст событие интеграции при создании нового заказа.

Оба варианта требуют обработчика событий. В обработчике событий есть бизнес-логика, необходимая для взаимодействия с системой и кодом и понимания ее ответов. Первый вариант использовался в примере с Lego, где обработчик событий обнаруживал само событие. Чтобы реализовать этот пример, необходимо добавить два раздела кода:

- создание сервиса, который обменивается информацией со складом;
- переопределение метода `SaveChanges` (и `SaveChangesAsync`), чтобы добавить код для создания события интеграции и его ответа.

12.5.1 Создание сервиса, который обменивается данными со складом

Вы знаете, что события интеграции пересекают границы в приложении. Например, в случае с примером с Lego предполагается, что сайт, на котором клиенты размещают заказы, является отдельным от склада, что означает некую форму обмена данными, возможно, с помощью REST-совместимого API. В этом случае нужно создать класс, который обменивается данными с нужным складом и возвращает либо указание на успешное завершение, либо серию ошибок. Следующий листинг – один из способов реализовать код, который взаимодействует с внешним складом.

Листинг 12.10 Обработчик событий Warehouse, который обнаруживает и обрабатывает событие

```
public class WarehouseEventHandler : IWarehouseEventHandler
{
    private Order _order;

    public bool NeedsCallToWarehouse(DbContext context)
    {
        var newOrders = context.ChangeTracker
            .Entries<Order>()
            .Where(x => x.State == EntityState.Added)
            .Select(x => x.Entity)
            .ToList();

        if (newOrders.Count > 1)
            throw new Exception(
                "Can only process one Order at a time");

        if (!newOrders.Any())
            return false;

        _order = newOrders.Single();
        return true;
    }

    public List<string> AllocateOrderAndDispatch()
    {
        var errors = new List<string>();

        //... Код для обмена данными со складом;

        return errors;
    }
}
```

Этот метод обнаруживает событие и возвращает true, если есть заказ для отправки на склад

Получает все вновь созданные заказы

Бизнес-логика обрабатывает только один заказ на один вызов метода SaveChanges

Если нового заказа нет, возвращается false

Если есть заказ, сохраняет его и возвращает true

Этот метод будет обмениваться данными со складом и возвращать все ошибки, которые склад отправляет обратно

Добавляет код для связи со складом

Возвращает список ошибок. Если список пуст, код был успешным

12.5.2 Переопределение метода SaveChanges для обработки события интеграции

Как указывалось ранее, мы используем реализацию события интеграции, которая обнаруживает само событие, вместо того чтобы добавлять событие в класс сущности. Поэтому код внутри переопределенных методов SaveChanges и SaveChangesAsync относится к событию интеграции. В следующем листинге показан код для реализации примера с Lego.

Листинг 12.11 DbContext с переопределенным методом SaveChanges и обработчиком событий Warehouse

```
public class IntegrationEventDbContext : DbContext
{
    private readonly IWarehouseEventHandler
        _warehouseEventHandler;
```

Содержит экземпляр кода, который будет обмениваться данными с внешним хранилищем

```
public IntegrationEventDbContext(
    DbContextOptions<IntegrationEventDbContext> options,
    IWarehouseEventHandler warehouseEventHandler)
    : base(options)
{
    _warehouseEventHandler = warehouseEventHandler;
}

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

public override int SaveChanges
    (bool acceptAllChangesOnSuccess)
{
    if (!_warehouseEventHandler.NeedsCallToWarehouse(this))
        return
            base.SaveChanges(acceptAllChangesOnSuccess);

    using(var transaction = Database.BeginTransaction())
    {
        var result =
            base.SaveChanges(acceptAllChangesOnSuccess);

        var errors = _warehouseEventHandler
            .AllocateOrderAndDispatch();

        if (errors.Any())
        {
            throw new OutOfStockException(
                string.Join('.', errors));
        }

        transaction.Commit();
        return result;
    }
}

//… Переопределенный метод SaveChangesAsync не указан
}
```

Внедряет обработчик складских событий, используя внедрение зависимостей

Переопределяет метод SaveChanges для включения обработчика событий Warehouse

Если обработчик событий не обнаруживает события, выполняется обычный метод SaveChanges

Произошло событие интеграции, значит, открывается транзакция

Вызывает базовый метод SaveChange для сохранения заказа

Вызывает обработчик событий Warehouse, который обменивается данными со складом

Если склад вернул ошибки, выбрасывается исключение OutOfStockException

Если ошибок не было, заказ фиксируется в базе данных

Возвращает результат метода SaveChanges

ПРИМЕЧАНИЕ Когда вы используете транзакции, в которых активирована опция повторной попытки при сбое, необходимо обернуть транзакцию в стратегию выполнения (см. раздел 11.7.1).

12.6 Улучшение события предметной области и реализаций событий интеграции

Показанный до сих пор код реализует полностью работающую систему событий предметной области и событий интеграции, кото-

рую можно использовать, но в ней отсутствуют некоторые полезные функции. Например, важно добавить асинхронные обработчики событий. В этом разделе мы изучим дополнительные возможности, которые можно было бы добавить к обработке событий. Вот некоторые дополнительные возможности, которые я обнаружил, когда создавал универсальную библиотеку событий `EfCore.GenericEventRunner`:

- обобщение событий (события, происходящие до, во время и после вызова метода `SaveChanges`);
- добавление поддержки асинхронных обработчиков событий;
- выполнение нескольких обработчиков для одного и того же события;
- выполнение последовательности событий, в которых одно событие запускает другое.

В следующих разделах мы добавим возможности из этого списка к вариантам с событиями предметной области и интеграции, над которыми работали до сих пор. Цель состоит в том, чтобы создать универсальную библиотеку, которую можно использовать в любом приложении, где события могут оказаться полезными.

ПРИМЕЧАНИЕ Из-за нехватки места полная реализация нововведений не предоставляется. Цель состоит в том, чтобы показать, какие улучшения можно было бы добавить в средство обработки заказов. Ссылки на реализацию в библиотеке `EfCore.GenericEventRunner` предоставляются там, где это необходимо.

12.6.1 Обобщение событий: запуск до, во время и после вызова метода SaveChanges

Если вы собираетесь создать библиотеку для обработки событий, то стоит рассмотреть все типы событий, которые вы, возможно, захотите обработать. Вы уже видели события предметной области и события интеграции, но в примере с событием интеграции мы писали систему событий интеграции вручную, потому что так было проще. Однако если вы хотите написать библиотеку, то стоит приложить усилия для обработки событий интеграции.

Может быть полезен еще один тип событий – тот, что запускается, когда методы `SaveChanges` или `SaveChangesAsync` успешно завершили работу. Вы можете отправить электронное письмо, если уверены, что заказ был проверен и успешно добавлен в базу данных. В этом примере используются три типа событий, которые я называю событиями *До* (события предметной области), *Во время* (события интеграции) и *После* (рис. 12.7).

Для реализации системы событий «До, во время и после» необходимо добавить еще два диспетчера событий (см. листинг 12.5): один вызывается внутри транзакции для обработки события интеграции, а второй – после успешного завершения работы метода `SaveChanges`

или SaveChangesAsync (рис. 12.7). Кроме того, вам понадобятся три интерфейса для обработчиков событий: до, во время и после, чтобы в каждый момент вызывался правильный обработчик событий.

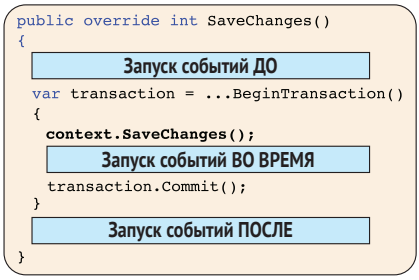


Рис. 12.7 Изучение различных событий, связанных с вызовом метода SaveChanges или SaveChangesAsync, предлагает три важные позиции: 1) до вызова метода SaveChanges, что позволяет изменять сущности перед тем, как их сохранить; 2) внутри транзакции, где был вызван этот метод, но транзакция не была зафиксирована, позволяя откатить сохраненные данные в случае сбоя исходящего события; и 3) после того как методы SaveChanges или SaveChangesAsync успешно завершили работу, что позволяет запустить код, который действителен только в том случае, если данные были успешно сохранены

Полная реализация довольно длинная и здесь не приводится. Код из метода RunEventsBeforeDuringAfterSaveChanges, который реализует систему событий «До, во время и после», можно найти в библиотеке EfCore.GenericEventRunner: http://mng.bz/K4A0.

12.6.2 Добавление поддержки асинхронных обработчиков событий

Во многих современных многопользовательских приложениях асинхронные методы улучшают масштабируемость, поэтому нам нужны асинхронные версии обработчиков событий. Добавление асинхронного метода требует дополнительного интерфейса обработчика событий. Кроме того, чтобы найти асинхронную версию обработчика при вызове метода SaveChangesAsync, необходимо изменить код диспетчера событий. В листинге 12.12 показан обновленный метод RunEvents в диспетчере событий из листинга 12.5, который превращается в асинхронный метод RunEventsAsync.

ПРИМЕЧАНИЕ Чтобы помочь вам увидеть изменения, добавленные к версии из листинга 12.5, я добавил комментарии только к измененному коду.

Листинг 12.12 Исходный метод RunEvents обновлен для запуска асинхронных обработчиков событий

```
public async Task RunEventsAsync(DbContext context)
{
    var allEvents = context.
        ChangeTracker.Entries<IEntityEvents>()
        .SelectMany(x => x.Entity.GetEventsThenClear());

    foreach (var domainEvent in allEvents)
    {
        var domainEventType = domainEvent.GetType();
        var eventHandleType = typeof(IEventHandlerAsync<>)
            .MakeGenericType(domainEventType);

        var eventHandler =
            _serviceProvider.GetService(eventHandleType);
        if (eventHandler == null)
            throw new InvalidOperationException(
                "Could not find an event handler");

        var handlerRunnerType =
            typeof(EventHandlerRunnerAsync<>)
            .MakeGenericType(domainEventType);
        var handlerRunner = ((EventHandlerRunnerAsync)
            Activator.CreateInstance(
            handlerRunnerType, eventHandler));

        await handlerRunner.HandleEventAsync(domainEvent);
    }
}
```

RunEvent становится асинхронным методом, а его имя меняется на RunEventAsync

Теперь код ищет дескриптор с асинхронным типом

Требуется асинхронный EventHandlerRunner для запуска обработчика событий

Приводится к асинхронному методу

Разрешает коду запускать асинхронный обработчик событий

12.6.3 Несколько обработчиков событий для одного и того же события

Для события можно определить несколько обработчиков событий. Например, у события `LocationChangedEvent` может быть один обработчик событий для пересчета налогов и еще один обработчик для обновления карты текущих проектов компании. В текущей реализации диспетчеров событий метод внедрения зависимостей `GetService` выбросит исключение, потому что может вернуть только один сервис. Решение простое. Используйте метод `GetServices`, а затем переберите каждый найденный обработчик событий:

```
var eventHandlers =
    _serviceProvider.GetServices(eventHandleType);
if (!eventHandlers.Any())
    throw new InvalidOperationException(
            "Could not find an event handler");
```

```
foreach(var eventHandler in eventHandlers)
{
    //… Используйте код из листинга 12.5, который запускает единственный
      обработчик событий;
```

12.6.4 Последовательности событий, в которых одно событие запускает другое

В системе моего клиента мы обнаружили, что одно событие может привести к созданию нового. Событие `LocationChangedEvent` обновляло `SalesTax`, что, в свою очередь, вызывало событие `QuotePriceChangeEvent`. Эти обновления называются *последовательностями событий*, потому что бизнес-логика состоит из серии шагов, которые должны выполняться в определенном порядке для завершения бизнес-правила.

Обработка последовательности событий требует, чтобы вы добавили схему циклов, которая ищет события, создаваемые другими событиями. В следующем листинге показан обновленный метод `RunEvents` в диспетчере событий из листинга 12.5, с комментариями только в новом коде цикла.

Листинг 12.13 Добавление цикла по событиям к методу `RunEvents` в диспетчере событий

```
public void RunEvents(DbContext context)
{
    bool shouldRunAgain;
    int loopCount = 1;
    do
    {
        var allEvents = context.
            ChangeTracker.Entries<IEntityEvents>()
            .SelectMany(x => x.Entity.GetEventsThenClear());

        shouldRunAgain = false;
        foreach (var domainEvent in allEvents)
        {
            shouldRunAgain = true;

            var domainEventType = domainEvent.GetType();
            var eventHandleType = typeof(IEventHandler<>)
                .MakeGenericType(domainEventType);
            var eventHandler =
                _serviceProvider.GetService(eventHandleType);
            if (eventHandler == null)
                throw new InvalidOperationException(
                    "Could not find an event handler");

            var handlerRunnerType = typeof(EventHandlerRunner<>)
                .MakeGenericType(domainEventType);
```

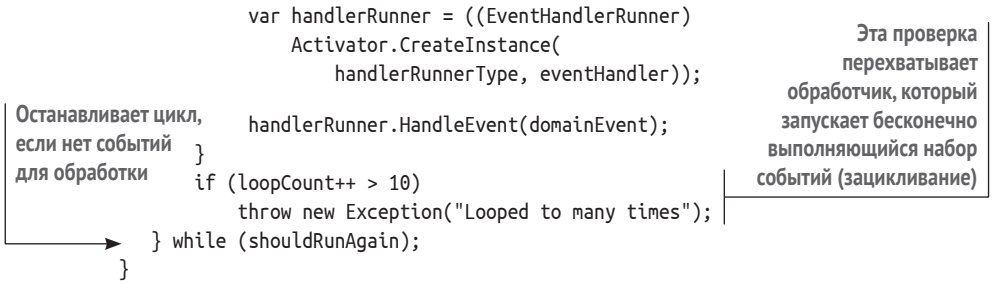
Контролирует, когда необходимо выполнить выход из цикла

Подсчитывает, сколько раз диспетчер событий проверяет наличие других событий

Код в цикле do/while выполняется, пока переменная shouldRunAgain имеет значение true

Переменной shouldRunAgain присваивается значение false. Если событий нет, то произойдет выход из цикла

Если есть события, то переменной shouldRunAgain будет присвоено значение true



Резюме

- Класс события предметной области несет сообщение, которое хранится внутри класса сущности. Это событие определяет тип события и несет в себе данные, относящиеся к конкретному событию, например какие данные изменились.
- Обработчики события содержат бизнес-логику, предназначенную для события предметной области. Их работа заключается в запуске бизнес-логики, используя данные событий предметной области для управления ее действиями.
- Версия событий предметной области с методами `SaveChanges` и `SaveChangesAsync` перехватывает все события предметной области в классах отслеживаемых сущностей, а затем запускает соответствующие обработчики событий.
- Версии событий интеграции с методами `SaveChanges` и `SaveChangesAsync` используют транзакцию, чтобы гарантировать, что и база данных, и обработчик события интеграции успешно завершат работу до обновления базы данных. Это требование позволяет синхронизировать две отдельные части приложения.
- В разделе 12.4 мы реализовали систему событий предметной области, создав классы событий предметной области, обработчики событий и диспетчер событий. Используя эти три части и переопределяя методы `SaveChanges` и `SaveChangesAsync`, можно применять события предметной области в своих приложениях.
- В разделе 12.5 мы обновили систему событий предметной области из раздела 12.4 для обработки событий интеграции, которая требует вызова внешнего сервиса в транзакции базы данных.
- В разделе 12.5 мы добавили улучшения в диспетчер событий, такие как поддержка обработчиков событий, использующих асинхронные методы.

13 Предметно-ориентированное проектирование и другие архитектурные подходы

В этой главе рассматриваются следующие темы:

- три архитектурных подхода, примененных к приложению Book App из части III;
- различия между обычными классами сущностей и классами сущностей в стиле предметно-ориентированного проектирования;
- восемь способов применения предметно-ориентированного проектирования к классам сущностей;
- три способа решения проблем, связанных с производительностью при использовании предметно-ориентированного проектирования.

Хотя эта книга посвящена EF Core, я хочу добавить кое-что об архитектуре программного обеспечения, поскольку читатели первого издания данной книги сочли эту тему полезной. В первой части вы познакомились с многоуровневой архитектурой. Теперь в третьей части, в которой мы создаем гораздо более сложное приложение Book App, я изменю его архитектуру, чтобы улучшить разделение частей кода и сделать данные классов сущностей более безопасными.

Самым важным из этих архитектурных изменений является переход на использование предметно-ориентированного проектирова-

ния (Domain Driven Design – DDD) из одноименной книги Эрика Эванса (Addison Wesley Professional, 2003 г.). В первой версии EF Core было добавлено одно нововведение, которого не было в EF6, – резервные поля, – благодаря чему стало возможным использовать DDD. С момента выхода первого издания этой книги я часто использовал DDD как в клиентских приложениях, так и при создании библиотек для обработки классов сущностей.

Я поделюсь своим опытом и кодом, чтобы помочь вам узнать, как DDD может помочь вам при разработке приложений. Использование DDD в классах сущностей разбито на восемь разделов, чтобы вы могли понять, как каждая часть помогает улучшить приложение. В конце я расскажу о том, как справиться с низкой производительностью обновлений, когда у вас много записей в связи, использующей сущности в стиле DDD.

13.1 Хорошая программная архитектура упрощает создание и сопровождение приложения

Одна из проблем при создании приложений заключается в том, что по мере роста их становится труднее разрабатывать, потому что нужно изменять существующий код, чтобы добавить что-то новое. Возникают всевозможные проблемы, например поиск и понимание существующего кода, принятие решения, как лучше всего добавить это нововведение и убедиться, что вы ничего не сломали.

Архитектура, которую вы выбираете для своего приложения, – это один из способов упростить написание и обновление кода. Принципы разработки программного обеспечения, такие как разделение ответственностей и DDD, также играют свою роль в упрощении исправлений и расширения приложения. Кроме того, хороший дизайн приложения предоставляет паттерн, который поможет вам при написании кода, а также правила, которые подтолкнут вас на правильный путь разработки.

В своей книге «Эволюционная архитектура. Поддержка непрерывных изменений» (O'Reilly, 2017) Нил Форд ввел термин «эволюционная архитектура», отражающий тот факт, что в настоящее время приложениям необходимо расти и меняться, чтобы по-прежнему предоставлять пользователю нужные возможности и лучший опыт. В разделе 13.2 я описываю принципы архитектуры и программного обеспечения, которые выбрал для приложения Book App из третьей части: принципы, которые значительно упрощают добавление функциональности.

13.2 Развивающаяся архитектура приложения Book App

В первой и второй частях книги приложение Book App использует одну базу данных, содержащую около 50 книг. Цель – разработать простое приложение, чтобы показать, как использовать различные функции EF Core в реальном приложении. Следовательно, многоуровневая архитектура (см. раздел 5.2) – подходящий вариант.

Приложение Book App из первой и второй частей достаточно небольшое, и я мог бы поместить весь код в приложение ASP.NET Core, но я этого не сделал. Вместо этого я использовал многоуровневую архитектуру, где бóльшая часть интересного кода находится на уровне данных и сервисном уровне. И вот почему:

- было бы сложнее найти что-то внутри единого проекта ASP.NET Core;
- было бы сложнее провести тестирование, потому что код был бы вшит в проект с ASP.NET Core.

Помимо многоуровневой архитектуры, я использовал программный принцип разделения ответственностей (см. раздел 2.7), чтобы разбить приложение на более мелкие части, потому что я знал, что буду добавлять функциональность. Вот два из множества примеров, показывающие, почему принцип разделения ответственностей так полезен:

- я строю основной запрос приложения Book App по частям (Выборка, Фильтрация, Сортировка, Разбиение на страницы), чтобы его легче было понять, протестировать и провести рефакторинг;
- во второй части я переместил код Fluent API в классы настройки по отдельным сущностям, чтобы упростить поиск, отображение и рефакторинг конфигурации для конкретного класса сущности.

Многоуровневая архитектура хорошо работает в первой и второй частях, где основное внимание уделяется тому, как работает EF Core, но третья часть посвящена настройке производительности приложений EF Core. В этой части используется несколько баз данных (SQL и Cosmos DB), два способа доступа к базе данных (EF Core и Dapper), а также несколько методов оптимизации производительности. Это означает, что существуют разные версии кода для отображения книг в базе данных приложения Book App. Чтобы управлять всеми этими подходами к запросам и показать вам новые способы разработки приложений, я использую три новых архитектурных/программных принципа для создания приложения Book App в третьей части:

- *модульный монолитный* подход, реализующий принцип разделения ответственностей, используя проекты .NET;
- принципы DDD (предметно-ориентированного проектирования) как в архитектуре, так и в классах сущностей;
- *чистая архитектура*, согласно описанию Роберта С. Мартина (Дяди Боба).

Я даю введение в эти три принципа в разделах с 13.2.1 по 13.2.3. На рис. 13.1 представлен общий вид архитектуры приложения Book App из третьей части.

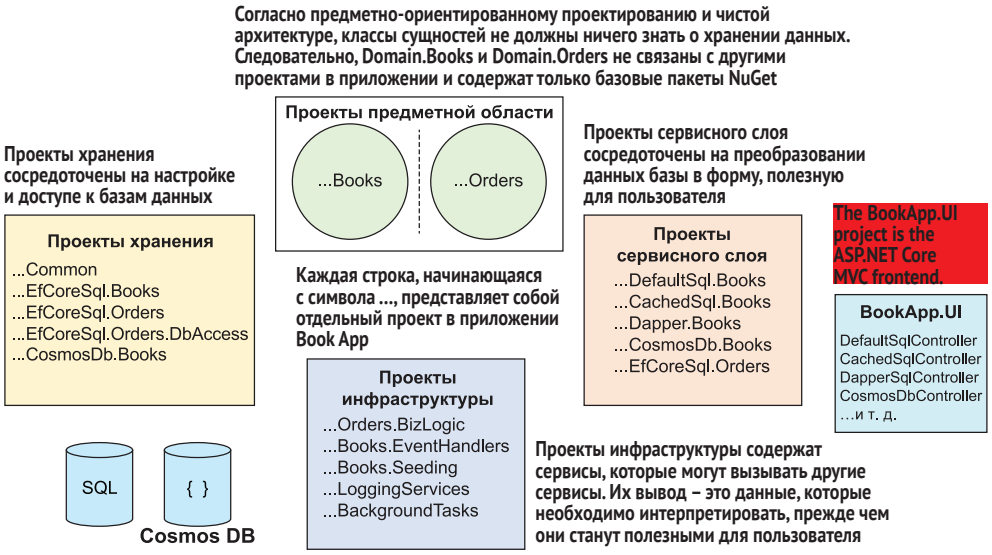


Рис. 13.1 Обзор того, как создано приложение Book App из третьей части, с пятью группами: Предметная область, Хранение, Инфраструктура, Сервисный слой и Пользовательский интерфейс на ASP.NET Core. Эта структура предназначена для обеспечения соблюдения правил принципа разделения ответственностей, ограничивая доступ разработчика. Цель состоит в том, чтобы разбить код на отдельные «функции», дабы упростить понимание и рефакторинг кода

13.2.1 Создание модульного монолита для обеспечения реализации принципов разделения ответственностей

Моя цель – сделать приложение модульным. Говоря *модульным,* я подразумеваю, что код для конкретной возможности, например отображения книг, легко идентифицировать и у него нет ссылок на код функциональности, который ему не нужен. Я добиваюсь этой цели, создавая небольшие проекты, которые реализуют код для конкретной задачи и ссылаются только на проекты с кодом, необходимым для нее (см. рис. 13.2).

У многоуровневой архитектуры в первой и второй частях есть проект «Сервисный уровень» (ServiceLayer), содержащий код множества функциональных возможностей приложения Book App, включая код для отображения книг, создания заказа, заполнения базы данных и запуска фоновых сервисов. Такая архитектура превращается в массу сильносвязанного кода (известного как *комок грязи*), и его рефакторинг сложно осуществлять. Используя модульный монолит в третьей части, каждая часть функциональности выделена в отдельный про-

ект, что (почти) не позволяет им совместно использовать код, кроме как через нижний уровень.

На рис. 13.1 это неочевидно, но разные проекты связаны друг с другом для создания функциональных возможностей, которые являются максимально автономными. На рис. 13.2 показаны две функции, одна для обработки книг, а вторая для обработки заказов от пользователей. Они являются независимыми (помимо проекта Persistence.Common), и у них только один общий проект.

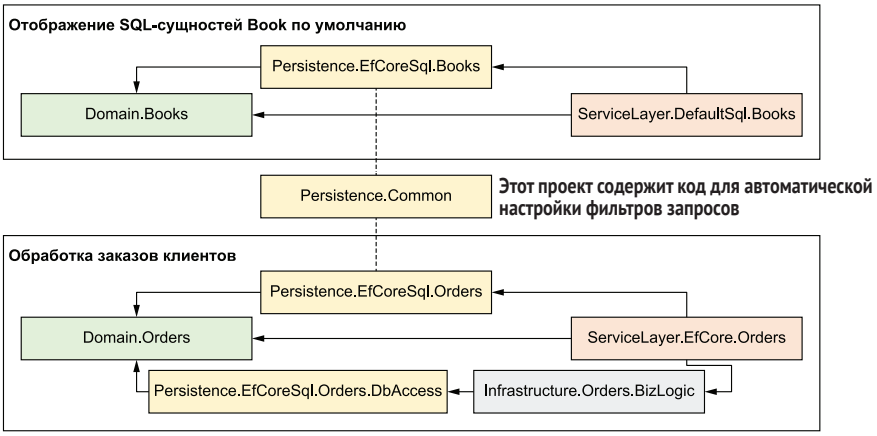


Рис. 13.2 Модульный монолит следует принципу разделения ответственностей, разбивая код на небольшие проекты, каждый из которых выполняет определенную работу. Кроме того, этот подход следует чистой архитектуре с четырьмя уровнями: Предметная область, Хранение, Инфраструктура и Сервисный уровень, – как показано в названиях каждого проекта. Некоторые имена проектов, например ServiceLayer.DefaultSql.Books, сейчас, возможно, не имеют смысла, но они станут понятны, когда вы дойдете до глав 15 и 16

Основная цель применения модульного подхода к монолиту – преодолеть типичную структуру «комка грязи». Для лучшего понимания можно рассматривать функциональные возможности в модульной монолитной архитектуре как микросервисы, которые обмениваются данными посредством простых вызовов методов, тогда как микросервисы используют для этой цели некий протокол с возможными сбоями.

ОПРЕДЕЛЕНИЕ *Архитектура микросервисов* организует приложение как коллекцию слабосвязанных сервисов, которые обмениваются данными с помощью некой формы передачи сообщений, например HTTP-сообщений.

Модульный подход дает множество преимуществ:

- можно легко найти весь код, связанный с определенной командой, например отображение книги;

- изменение функции не должно иметь никакого эффекта или должно оказывать минимальное влияние на другие функции;
- вы должны иметь возможность переместить функцию в другое приложение, например микросервис, с минимальными проблемами для остальной части приложения.

В то же время вы хотите, чтобы ваш код следовал принципу DRY (Don't repeat yourself – Не повторяйся), поэтому будет присутствовать некий общий код, например проект Persistence.Common, показанный на рис. 13.2. Но я не рекомендую создавать большое количество общего кода, который используется в большом количестве проектов, потому что изменение этого кода может нарушить код другой части приложения. Если ваш общий код настолько хорош, превратите его в библиотеку. (Так делаю я.)

В этом разделе мы завершаем обзор общей архитектуры приложения Book App. Остальная часть данной главы посвящена предметно-ориентированному проектированию, поскольку оно может существенно повлиять на то, как вы создаете и используете свой код EF Core и классы сущностей.

ПРИМЕЧАНИЕ Поскольку я внес значительные изменения в главы 15 и 16, я добавил новый раздел 13.8 в этой главе, чтобы поделиться своим опытом использования трех архитектурных подходов, поскольку добавил нововведения, которые удвоили размер приложения Book App начиная с главы 13.

13.2.2 Использование принципов предметно-ориентированного проектирования в архитектуре и в классах сущностей

Предметно-ориентированное проектирование (DDD) подробно описывает многие подходы к определению, созданию и управлению приложениями. Но я хочу особо выделить три принципа:

- классы сущностей в приложении Book App (часть III) следуют правилам DDD для того, что называется *сущностями* (а в EF Core – *классами сущностей*). Основное правило заключается в том, что сущность в DDD полностью контролирует данные в ней: все свойства – это свойства с доступом только на чтение, а для создания и обновления данных используются конструкторы и методы. Предоставляя сущности полный контроль над ее данными, вы значительно расширяете возможности классов сущностей; у каждого класса сущности есть четко определенные конструкторы/методы, которые может использовать разработчик;
- DDD гласит, что сущности, содержащие данные и (бизнес-)логику предметной области, не должны ничего знать о том, как сущности сохраняются в базе данных (об этом идет речь в раз-

деле 4.3.1). Я подробнее расскажу об этом в разделе 13.2.3, где говорится об использовании чистой архитектуры;

- в DDD говорится об *ограниченных контекстах*, которые разделяют ваше приложение на отдельные части. Идея состоит в том, чтобы создать отдельные ограниченные контексты, чтобы их легче было понять, а затем настроить четко определенный обмен данными между ними. В приложении Book App для третьей части я создал ограниченный контекст вокруг отображения и редактирования книг и еще один контекст для заказа книг.

13.2.3 Применение чистой архитектуры согласно описанию Роберта Мартина

Чистая архитектура – это подход к разработке программного обеспечения, разделяющий различные части кода на уровни, расположенные в виде ряда колец, как у лука. Эти уровни плюс некоторые правила нужны для того, чтобы организовать код так, чтобы классы сущностей и бизнес-логика были изолированы от более высоких уровней в разных «кольцах». Я не смог поместить все проекты в рис. 13.1, используя серию колец, но приложение Book App все же следует чистой архитектуре.

> **ПРИМЕЧАНИЕ** Вот ссылка на определение чистой архитектуры, написанное Робертом С. Мартином (Дядей Бобом): http://mng.bz/9N71.

Чистая архитектура включает в себя несколько других архитектур, в том числе гексагональную и onion-архитектуру. Цель данной архитектуры – установить правила, определяющие, как разные уровни обмениваются данными. Например, в чистой архитектуре есть правило зависимости, утверждающее, что код на внутренних кольцах не может явно связываться с внешними кольцами. Чистая архитектура соответствует правилу DDD, призывающему отделять сущности от кода базы данных (хранение), и помогает разделить код на кольца, которые я определил как Предметная область, Хранение, Инфраструктура, Сервисный уровень и Пользовательский интерфейс на ASP.NET Core.

13.3 Введение в предметно-ориентированное проектирование на уровне класса сущности

DDD – это обширная и многогранная тема, но эта книга посвящена EF Core. Поэтому я сосредоточусь на классах сущностей в EF Core: отделении взаимодействия с базой данных от сущностей DDD и исполь-

зовании паттерна «Ограниченный контекст» для определения того, как код обращается к базе данных.

Книга Эрика Эванса «*Предметно-ориентированное проектирование*», где основное внимание уделяется паттернам и приемам проектирования, которые делают создание приложений лучше и актуальнее, является ключевой при разработке программного обеспечения. Чего в ней нет, так это подробного набора шагов по реализации DDD. Я думаю, это хорошо, потому что если бы в книге были даны подробные инструкции, к настоящему времени она уже стала бы неактуальной.

Однако, поскольку в книге Эванса не было подробных планов реализации, многие придумали разные способы реализации DDD. С одной стороны, классы сущностей тщательно проработаны, чтобы содержать только бизнес-логику; все части базы данных, например первичный и внешний ключи, скрыты. С другой – конструкции, в которых – из-за желания разработчика поместить весь код бизнес-логики внутри класса сущности – класс сущности содержит операции чтения и записи в базу данных. Я опишу подход, который применяет большинство разработчиков, плюс некий вариант для сокращения кода, который вы должны писать. (Я не буду показывать строгий стиль, когда все ключи скрыты, но можно следовать этому подходу, используя теневые свойства.)

Однако для начала рассмотрим основные различия между обычным классом сущности и классом сущности DDD, что поможет вам понять разницу между тем, что вы уже видели в этой книге, и тем, как работает DDD. Начнем с простого обновления свойства `PublishedOn` сущности `Book` в качестве примера обновления базы данных; впервые вы увидели это обновление в разделе 3.3. Это тривиальный код, позволяющий увидеть различия в двух подходах. На рис. 13.3 слева показан исходный вариант, не использующий DDD, а справа – вариант с DDD.

Версия с DDD на рис. 13.3 требует немного больше кода, но, как вы увидите в разделе 13.4, этот дополнительный код позволяет классам сущностей стать намного более ценными. Тем не менее, когда потенциально у вас сотни операций создания и обновления, эти несколько лишних строк добавляют работы, поэтому я всегда пытаюсь найти способы уменьшить код, который мне нужно писать.

13.4 Изменение сущностей приложения Book App, чтобы следовать предметно-ориентированному проектированию

В этом разделе мы изменим класс сущности `Book` и связанные классы сущностей, чтобы следовать DDD. Мы будем вносить изменения по-

этапно, чтобы понять, как и зачем это нужно. Вот этапы изменения нашего кода на DDD-подход:

- изменение свойств сущности Book на доступ только для чтения;
- обновление свойств сущности Book с помощью методов в классе сущности;
- управление созданием сущности Book;
- разбор различий между сущностью и объектом значения;
- минимизация связей между классами сущностей;
- группировка классов сущностей (в DDD это *агрегаты*);
- принятие решения относительно того, когда бизнес-логику не следует запускать внутри сущности;
- применение паттерна «Ограниченный контекст» к DbContext.

13.4.1 Изменение свойств сущности Book на доступ только для чтения

Согласно DDD, класс сущности отвечает за данные, которые он содержит; следовательно, он должен контролировать, как данные создаются или изменяются. Чтобы класс сущности мог управлять своими данными, мы сделаем так, чтобы все свойства сущности были свойствами с доступом только на чтение. После этого разработчик может установить данные в классе сущности только через конструктор класса (раздел 13.4.3) или используя методы класса сущности (раздел 13.4.2). Сущность должна гарантировать, что всегда находится в допустимом состоянии. В случае с классом сущности Book, где не используется DDD, я мог бы создать книгу без автора, но бизнес-правила гласят, что у действующей книги должен быть хотя бы один автор. Чтобы получить такой уровень контроля, нужно сделать так, чтобы все свойства были с доступом только на чтение, дабы разработчик использовал определенные методы или конструкторы. В листинге 13.1 показан класс сущности Book со свойствами с доступом только на чтение.

Листинг 13.1 Делаем так, чтобы свойства класса сущности Book были с доступом только на чтение

```
public class Book
{
    public int BookId { get; private set; }
    public string Title { get; private set; }
    //… Остальные свойства, не относящиеся к коллекции, не указаны;

    private HashSet<Review> _reviews;
    public IReadOnlyCollection<Review>
        Reviews => _reviews?.ToList();

    private HashSet<BookAuthor> _authorsLink;
    public IReadOnlyCollection<BookAuthor>
        AuthorsLink => _authorsLink?.ToList();
```

У свойств, не относящихся к коллекции, методы записи являются закрытыми

Коллекция хранится в резервном поле

Коллекция свойств возвращает соответствующие резервные поля как коллекции с доступом только на чтение

```
    //… Остальные свойства, не относящиеся к коллекции, не указаны;
}
```

ПРЕДУПРЕЖДЕНИЕ Если вы используете AutoMapper, то он проигнорирует модификатор доступа `private` у метода записи и обновит свойство, а это не то, что нам нужно, когда мы используем DDD. Чтобы остановить это обновление, нужно добавить метод `IgnoreAllPropertiesWithAnInaccessibleSetter` после вызова метода AutoMapper `CreateMap<TSource,TDestination>`.

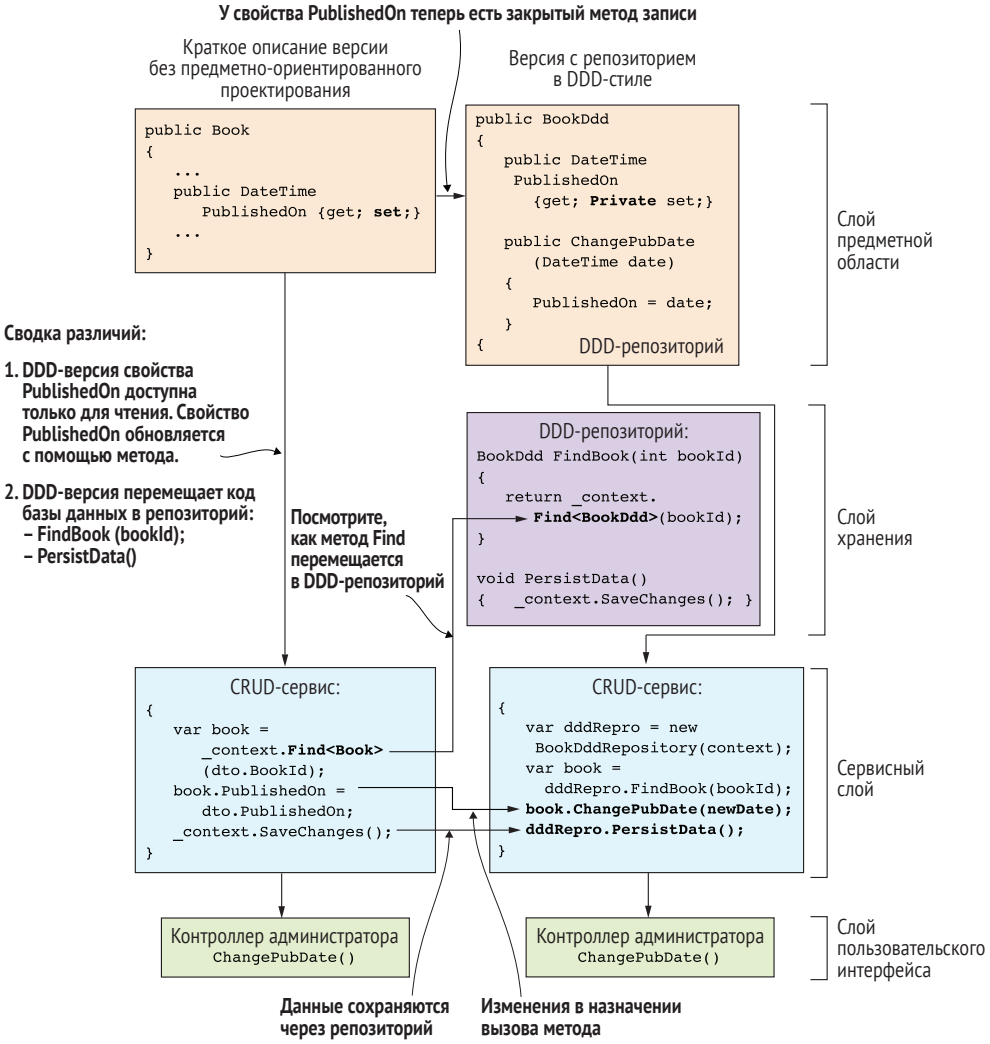


Рис. 13.3 Сравнение варианта без DDD для обновления даты публикации `Book` в приложении Book App (слева) с версией с DDD (справа). Код, необходимый для обновления, состоит из тех же частей, но версия с DDD перемещает весь код EF Core на уровень хранения. Кроме того, если вы «скрываете» DbContext в версии с DDD, то можете гарантировать, что разработчик может получить доступ к базе данных только через репозиторий DDD

13.4.2 Обновление свойств сущности Book с помощью методов в классе сущности

Когда все свойства преобразованы в свойства с доступом только на чтение, нужен другой способ обновить данные внутри сущности. Решение: добавить методы класса сущности, которые могут обновить свойства. Я называю их *методами доступа*. Создание методов доступа – дополнительная работа, так почему же DDD призывает это сделать? Вот основные преимущества:

- можно использовать такую сущность, как черный ящик. Методы доступа и конструкторы – это его API: сущность должна убедиться, что данные внутри нее всегда в допустимом состоянии;
- можно поместить свои бизнес-правила в метод доступа. Метод может возвращать ошибки пользователям, чтобы они могли исправить проблему и повторить попытку, или в случае проблемы в программе можно выбросить исключение;
- если нет способа обновить конкретное свойство, вы знаете, что вам запрещено изменять его.

Некоторые простые методы изменяют только свойство, но многие содержат бизнес-правила вашего приложения. Один из примеров в сущности `Book` – добавление и удаление промоакции. В классе сущности `Book` для третьей части мы заменяем класс сущности `PriceOffer` двумя методами, которые выполняют бизнес-правила для добавления и удаления этой промоакции. Вот правила:

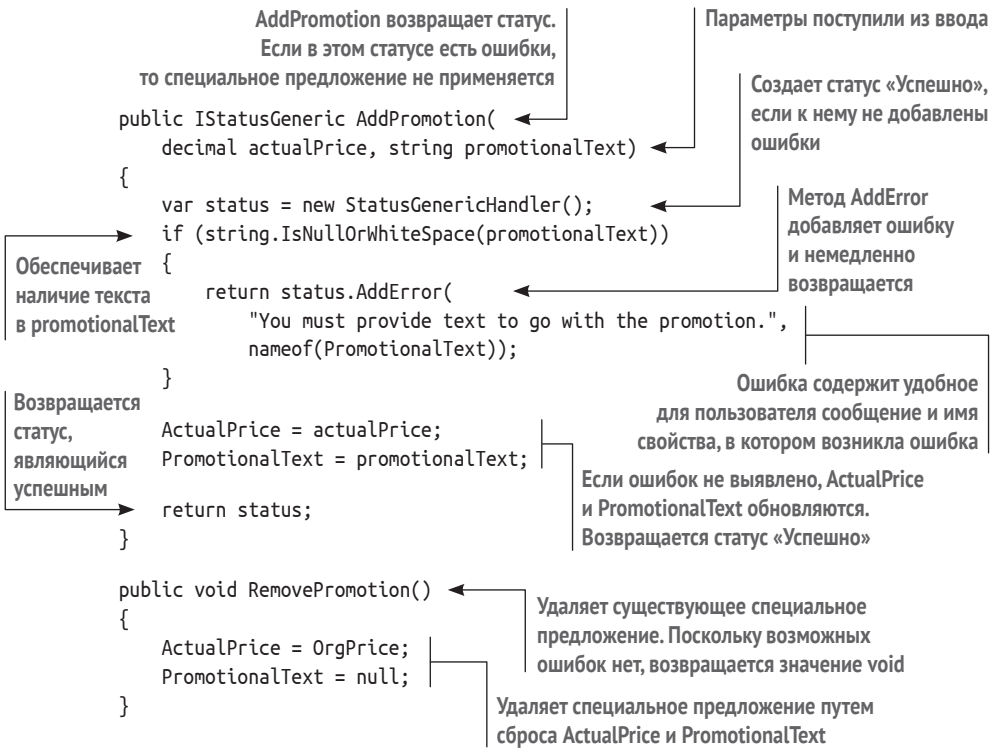
- цена, по которой продается книга, указана в свойстве `ActualPrice`;
- полная стоимость книги указана в свойстве `OrgPrice`;
- свойство `PromotionalText` должно иметь значение `null`, если промоакции нет, но если она есть, то сообщение должно присутствовать.

Было бы легко нарушить эти правила, но превращение их в метод доступа означает, что никто не сможет ошибиться. Кроме того, правила находятся в одном месте, чтобы их можно было легко поменять при необходимости. Эти методы доступа – одна из самых мощных техник DDD.

В листинге 13.2 показаны методы доступа `AddPromotion` и `RemovePromotion` в сущности `Book`. Они гарантируют, что правила добавления и удаления специальной цены соблюдаются.

ПРИМЕЧАНИЕ Интерфейс `IStatusGeneric` и класс `StatusGenericHandler` взяты из небольшой библиотеки NuGet с открытым исходным кодом, `GenericServices.StatusGeneric`, которую я использую во многих своих библиотеках и приложениях.

Листинг 13.2 Пример метода доступа, который содержит бизнес-логику и валидацию




```
public IStatusGeneric AddPromotion(
    decimal actualPrice, string promotionalText)
{
    var status = new StatusGenericHandler();
    if (string.IsNullOrWhiteSpace(promotionalText))
    {
        return status.AddError(
            "You must provide text to go with the promotion.",
            nameof(PromotionalText));
    }

    ActualPrice = actualPrice;
    PromotionalText = promotionalText;

    return status;
}

public void RemovePromotion()
{
    ActualPrice = OrgPrice;
    PromotionalText = null;
}
```

ПРИМЕЧАНИЕ Имя свойства в методе `AddError` из листинга 13.2: `PromotionalText`, а не `promotionalText`, потому что мы предоставляем имя свойства, которое использовалось в клиентской части ASP.NET Core при вызове метода `AddPromotion`.

13.4.3 Управление процессом создания сущности Book

В соответствии с DDD сущность контролирует установку данных в ней, поэтому нужно подумать о процессе ее создания. Насколько мне известно, Эрик Эванс не определяет этот процесс, но создание класса сущности – важный вопрос, особенно когда все свойства – с доступом только на чтение. Следовательно, необходимо предоставить хотя бы один конструктор или статический фабричный метод, который разработчик может использовать для создания нового экземпляра сущности.

Для класса сущности `Book` можно создать недопустимый экземпляр, поскольку бизнес-правила гласят, что свойство `Title` этой сущности не должно быть пустым и что должен быть по крайней мере один ав-

тор. Конструктор не может возвращать ошибки, поэтому мы создаем статический фабричный метод, который возвращает статус, содержащий ошибки, если название книги пустое или не указан автор. Если ошибок нет, то статус будет иметь свойство `Result`, содержащее только что созданную книгу, как показано в следующем листинге.

Листинг 13.3 Статический фабричный метод для создания допустимой книги или возврата ошибок

Создание приватного конструктора означает, что с помощью конструктора нельзя создать сущность

Статический метод CreateBook возвращает статус с допустимой книгой (при отсутствии ошибок)

```
private Book() { }

public static IStatusGeneric<Book> CreateBook(
    string title, DateTime publishedOn,
    decimal price,
    ICollection<Author> authors)
{
    var status = new StatusGenericHandler<Book>();
    if (string.IsNullOrWhiteSpace(title))
        status.AddError(
            "The book title cannot be empty.");

    var book = new Book
    {
        Title = title,
        PublishedOn = publishedOn,
        OrgPrice = price,
        ActualPrice = price,
    };
    if (authors == null)
        throw new ArgumentNullException(nameof(authors));

    byte order = 0;
    book._authorsLink = new HashSet<BookAuthor>(
        authors.Select(a =>
            new BookAuthor(book, a, order++)));

    if (!book._authorsLink.Any())
        status.AddError(
            "You must have at least one Author for a book.");

    return status.SetResult(book);
}
```

Эти параметры – все, что нужно для создания допустимой книги

Создает статус, который может возвращать результат, – в данном случае это Book

Добавляет ошибку. Обратите внимание на отсутствие быстрого возврата, чтобы можно было добавить другие ошибки

Устанавливает значения свойств

Параметр авторов, имеющий значение null, считается программной ошибкой и вызывает исключение

Создает класс BookAuthor в том порядке, в котором были указаны авторы. Если авторов нет, добавляем ошибку

Устанавливает для свойства Result статуса значение в виде нового экземпляра Book

Если есть ошибки, значение равно null

Для простых классов сущностей можно использовать открытый конструктор с определенными параметрами, но любые сущности, имеющие бизнес-правила и возвращающие сообщения об ошибках, должны использовать статический фабричный метод в классе сущности.

13.4.4 Разбор различий между сущностями и объектом-значением

В DDD говорится о *сущности* (например, сущность `Book`), но также говорится и об *объекте-значении*. Разница состоит в том, что однозначно определяет экземпляр каждого. Эрик Эванс говорит: «Идентификация сущностей – дело нужное», но «сделайте так, чтобы [объекты-значения] отражали смысл заложенных в него атрибутов [свойств]» (*Предметно-ориентированное проектирование,* стр. 98–99). Вот два примера, которые могут помочь:

- сущность не определяется данными внутри нее. Например, я ожидаю, что не один человек по имени Джон Смит писал книгу. Таким образом, для каждого автора по имени Джон Смит приложению Book App потребуется отдельная сущность `Author`;
- объект значения определяется данными внутри него. Если у меня есть адрес для отправки заказа и был создан еще один адрес с той же дорогой, городом, штатом, почтовым индексом и страной, то два экземпляра адреса считаются равными.

С точки зрения EF Core сущность DDD – это класс сущности EF Core, который сохраняется в базу данных с некой формой первичного ключа. Первичный ключ гарантирует, что сущность является уникальной в базе данных, и когда EF Core возвращает результаты запроса, включающего классы сущностей (и запрос не включает `AsNoTracking`), он использует единственный экземпляр для каждого класса сущности, у которого тот же первичный ключ (см. раздел 6.1.3).

Объект-значение можно реализовать, используя собственный тип EF Core (см. раздел 8.9.1). Основная форма собственного типа – это класс без первичного ключа; данные добавляются в таблицу, в которую он включен.

> **ПРИМЕЧАНИЕ** В приложении Book App нет объектов-значений, поэтому я не могу использовать его в качестве примера. Смотрите листинг 8.15, чтобы увидеть неплохой пример использования собственных типов в классе сущности.

13.4.5 Минимизация связей между классами сущностей

Эрик Эванс говорит: «Важно максимально ограничивать связи» (*Предметно-ориентированное проектирование*, стр. 83). Далее он говорит, что добавленные двусторонние связи между сущностями означают, что нужно понимать обе сущности при работе с одной из них, что затрудняет понимание кода. Его (и моя) рекомендация – свести связи к минимуму. Например, у `Book` есть навигационное свойство всех `Review` для `Book`, но у `Review` нет навигационного свойства для `Book` (см. раздел 8.2).

Свести к минимуму навигационные связи между классами сущностей легко. В разделе 8.2 я рассматриваю связи между классами сущностей `Book` и `Review`. Я пришел к выводу, что сущности `Book` необходима навигационная коллекция отзывов, связанных с ней, но сущность `Review` не нуждалась в навигационной связи с сущностью `Book`. Другими словами, для понимания сущности `Book` требуется некое представление о том, что делает сущность `Review`, но когда я имел дело с сущностью `Review`, мне нужно было понимать только то, что делает сущность `Review`.

13.4.6 Группировка классов сущностей

Еще один важный паттерн DDD, который называется *агрегаты*, предлагает рекомендации по работе со связанными сущностями. Принцип агрегатов гласит, что нужно группировать сущности, которые можно считать «одной единицей для изменения данных» (*Предметно-ориентированное проектирование*, стр. 126). Одна из сущностей в агрегате является *корневым объектом*, и любые изменения в других сущностях агрегата производятся через этот объект.

На рис. 13.4 показаны сущности-агрегаты вокруг DDD-версии класса сущности `Book`, используемого в приложении Book App. Любые изменения сущностей `Review` или `BookAuthor`, связанных с сущностью `Book`, можно изменить только с помощью методов доступа или конструкторов в сущности `Book`. Сущность `Author` находится за пределами агрегата `Book`, поскольку ее можно привязать к нескольким `Book`.

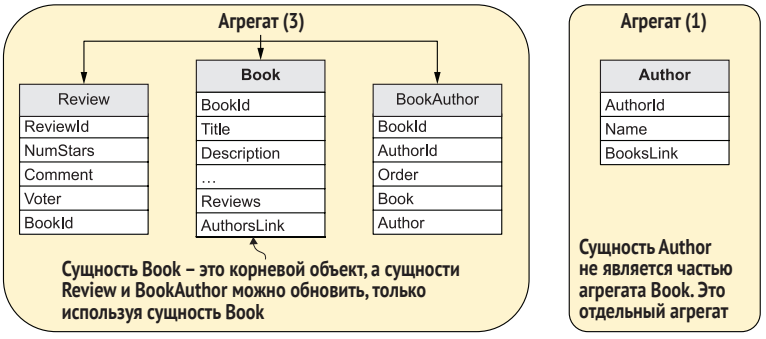


Рис. 13.4 Концепция агрегатов группирует сущности, которыми можно управлять так, как если бы они были одной группой данных. Одна из сущностей – это корневой объект (сущность `Book` слева и `Author` справа). Все обновления сущностей `Review` или `BookAuthor` производятся через сущность `Book`. Такой подход уменьшает количество сущностей, с которыми вам нужно иметь дело, и позволяет корневой сущности гарантировать, что все остальные агрегаты настроены правильно (например, что у `Book` есть хотя бы одна ссылка на `BookAuthor`)

ПРИМЕЧАНИЕ Сущность `BookAuthor` на рис. 13.4 нарушает правило агрегатов Эванса, потому что на некорневой объект нельзя ссылаться извне агрегата. (У сущности `Author` есть обратная ссылка на сущность `BookAuthor`.) Но сущность `BookAuthor` содержит данные, относящиеся к книге: свойство `Order`, которое определяет порядок упоминания авторов. Эти особенности `BookAuthor` делают ее агрегатом сущности `Book`.

Правило агрегатов упрощает обработку классов сущностей, поскольку одна корневая сущность может обрабатывать несколько агрегатов в своей группе. Кроме того, корневая сущность может подтвердить, что другие, некорневые агрегаты настроены правильно для корневого объекта, например фабричный метод `Book` проверяет наличие хотя бы одного автора для сущности `Book`.

Это правило также требует использования метода доступа в сущности `Book`, чтобы добавлять, обновлять или удалять ссылки сущностей `Review` для экземпляра объекта `Book`. В следующем листинге показаны два метода доступа для добавления и удаления отзывов.

Листинг 13.4 Методы доступа, управляющие агрегатом `Review`

```
public void AddReview(int numStars,                    // Добавляет новый отзыв с заданными параметрами
    string comment, string voterName)
{
    if (_reviews == null)                              // Этот код зависит от загружаемого поля _reviews, поэтому выдает исключение, если его значение равно null
        throw new InvalidOperationException(
            "Reviews collection not loaded");

    _reviews.Add(new Review(                           // Создает новый отзыв, используя его внутренний конструктор
        numStars, comment, voterName));
}

public void RemoveReview(int reviewId)                 // Удаляет отзыв, используя его первичный ключ
{
    if (_reviews == null)
        throw new InvalidOperationException(
            "Reviews collection not loaded");

    var localReview = _reviews.SingleOrDefault(        // Находит конкретный отзыв, который нужно удалить
        x => x.ReviewId == reviewId);

    if (localReview == null)                           // Отсутствие отзыва считается программной ошибкой, поэтому код выдает исключение
        throw new InvalidOperationException(
            "The review wasn't found");

    _reviews.Remove(localReview);                      // Найденный отзыв удален
}
```

Еще одно изменение, которое вы вносите, – это пометка конструктора класса сущности `Review` как `internal`. Это изменение не позволяет разработчику добавить отзыв путем создания экземпляра за пределами сущности `Book`.

13.4.7 Принимаем решение, когда бизнес-логику не следует помещать внутрь сущности

Согласно DDD, нужно поместить как можно больше бизнес-логики внутри сущностей, но правило агрегатов гласит, что корневой объект должен работать только с другими сущностями в группе агрегатов. Если у вас есть бизнес-логика, включающая несколько групп агрегатов, не следует помещать (всю) бизнес-логику в сущность; нужно создать внешний класс для реализации бизнес-логики.

Пример ситуации, требующей нескольких групп агрегатов в бизнес-логике, обрабатывает заказ пользователя на приобретение книг. Эта бизнес-логика включает в себя сущность `Book`, которая находится в группе агрегатов `Book/Review/BookAuthor`, и группу `Order/LineItem`.

Вы уже видели решение проблемы заказа книг в разделе 4.4.3. Версия с DDD использует аналогичный код, но заключительный этап создания `Order` выполняется в статическом фабричном методе внутри сущности `Order`, потому что `Order` – это корневой объект в группе агрегатов `Order/LineItem`. В следующем листинге показан внешний класс для реализации бизнес-логики `PlaceOrderBizLogic`.

> **ПРИМЕЧАНИЕ** Поскольку вы видели часть этого кода в листинге 4.2, здесь я опустил похожие части. Цель состоит в том, чтобы сосредоточить внимание на изменениях в частях с DDD, особенно касающихся создания `Order` с помощью статического фабричного метода `Order`.

Листинг 13.5 Класс `PlaceOrderBizLogic`, работающий с сущностями `Book` и `Order`

Этот метод возвращает статус с созданной сущностью Order, которая имеет значение null, если есть ошибки

PlaceOrderInDto содержит логическое значение TandC, а также коллекцию идентификаторов BookId и количество книг

```
public async Task<IStatusGeneric<Order>>
    CreateOrderAndSaveAsync(PlaceOrderInDto dto)
{
    var status = new StatusGenericHandler<Order>();

    if (!dto.AcceptTAndCs)
    {
        return status.AddError("accept T&Cs…");
    }
    if (!dto.LineItems.Any())
    {
        return status.AddError("No items in your basket.");
    }

    var booksDict = await _dbAccess
        .FindBooksByIdsAsync
            (dto.LineItems.Select(x => x.BookId));
```

Этот статус используется для сбора ошибок, и если ошибок нет, код возвращает заказ

Проверка пользовательского ввода

_dbAccess содержит код для поиска каждой книги (см. листинг 4.3)

Если при проверке каждой строки заказа были обнаружены ошибки, возвращается статус ошибки

Этот метод создает список bookId и количество книг (см. конец листинга 4.2)

```
var linesStatus = FormLineItemsWithErrorChecking
    (dto.LineItems, booksDict);
if (status.CombineStatuses(linesStatus).HasErrors)
    return status;

var orderStatus = Order.CreateOrder(
    dto.UserId, linesStatus.Result);
```

Вызывает статический фабричный метод Order. Его задача – сформировать заказ с помощью LineItems

Опять же, любые ошибки отменят заказ и вернут ошибки

```
if (status.CombineStatuses(orderStatus).HasErrors)
    return status;

await _dbAccess.AddAndSave(orderStatus.Result);
```

_dbAccess содержит код для добавления заказа и вызова метода SaveChangesAsync

```
return status.SetResult(orderStatus.Result);
}
```

Возвращает статус успешно с созданной сущностью Order

Самое большое изменение по сравнению с кодом из главы 4 состоит в том, что сущность `Order` берет на себя заключительный этап создания заказа. В следующем листинге показан статический фабричный метод `Order`.

Листинг 13.6 Этот статический фабричный метод создает `Order` с `LineItems` с проверкой ошибок

OrderBookDto находится в предметной области Order и несет информацию, необходимую Order

Этот статический фабричный метод создает заказ с набором заказанных позиций

Заказ использует UserId, чтобы показывать заказы только тому, кто их создал

```
public static IStatusGeneric<Order> CreateOrder
    (Guid userId,
    IEnumerable<OrderBookDto> bookOrders)
{
```

Создает статус для возврата с необязательным результатом Order

```
    var status = new StatusGenericHandler<Order>();
```

Устанавливает стандартные свойства заказа

```
    var order = new Order
    {
        UserId = userId,
        DateOrderedUtc = DateTime.UtcNow
    };

    byte lineNum = 1;
    order._lineItems = new HashSet<LineItem>(
        bookOrders
        .Select(x => new LineItem(x, lineNum++)));
```

Создает каждый из LineItems в порядке, в котором их добавил пользователь

```
    if (!order._lineItems.Any())
        status.AddError("No items in your basket.");
```

Повторно проверяет, действителен ли заказ

```
    return status.SetResult(order);
}
```

Возвращает статус с заказом. Если есть ошибки, статус задает для результата значение null

13.4.8 Применение паттерна «Ограниченный контекст» к DbContext приложения

В разделе 13.2.2 я сказал, что ограниченные контексты «разделяют ваше приложение на отдельные части» и что у этих контекстов есть «четко определенное взаимодействие». На рис. 13.1 вы видели два независимых проекта: `Persistence.EfCoreSql.Books` и `Persistence.EfCoreSql.Orders`. Но коду, описанному ранее для размещения заказа пользователя, требовалась информация о `Book`. Так как же справиться с этой ситуацией?

В данном конкретном случае решение состоит в использовании представления SQL в DbContext `Order`, которое отображается в таблицу Books в DbContext `Book`, как показано на рис. 13.5. Таким образом, можно сделать `Persistence.EfCoreSql.Books` и `Persistence.EfCoreSql.Orders` независимыми, и при этом у обоих будет доступ к данным в БД.

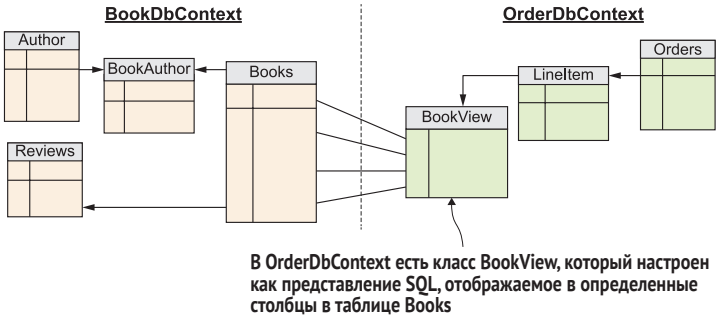


Рис. 13.5 Чтобы следовать подходу ограниченного контекста, Domain.Books должен быть независимым от Domain.Orders. Но на уровне базы данных им нужна сущность Book. Решением в этом случае является создание сущности `BookView` в Domain.Orders, которая содержит только определенные свойства для создания и отображения заказа. Затем мы настраиваем класс `BookView` как представление SQL, отображаемое в таблицу Books

Использование представления SQL – отличное решение, поскольку оно следует многим правилам DDD. Во-первых, `BookView` содержит только те данные, которые нужны стороне `Orders`, чтобы разработчик не отвлекался на нерелевантные данные. Во-вторых, когда класс сущности настроен как представление, EF Core помечает его как класс с доступом только на чтение, обеспечивая соблюдение правила DDD, согласно которому только сущность `Books` должна иметь возможность изменять данные в таблице Books.

ПРИМЕЧАНИЕ Еще одно преимущество заключается в том, что класс, отображаемый в представление SQL, не добавляет код миграции для изменения этой таблицы. Вы можете применить миграции EF Core как из контекстов `BookDbContext` и `Or-`

derDbContext к базе данных, и только сущность BookDbContext будет влиять на таблицу Books (см. раздел 9.4.3).

Хотя представление SQL хорошо подходит для этой цели, оно создает связь между двумя ограниченными контекстами. Будьте осторожны, если вы измените сущность Book, а затем выполните миграцию базы данных.

ПРИМЕЧАНИЕ Передача данных между ограниченными контекстами – большая тема, которую я не могу охватить здесь. Я рекомендую старую, но все еще актуальную статью «Стратегии интеграции ограниченных контекстов», в которой Филип Браун дает хороший обзор множества способов обмена данными между ограниченными контекстами (http://mng.bz/96Bg).

13.5 Использование классов сущностей в стиле DDD в вашем приложении

Цель DDD заключается в том, чтобы сосредоточить внимание на предметной области, т. е. сущностях и их связях. И наоборот, нам не важны части, касающиеся базы данных (хранение), которые отвлекают разработчика, работающего над проектированием предметной области. Идея состоит в том, что сущность и ее связи (навигационные свойства на языке EF Core) – это все, что должно волновать разработчика при решении проблем предметной области.

После обновления дизайна классов сущностей в соответствии с DDD нужно использовать эти классы в своем приложении. Запрос на получение книги не изменился, изменился способ создания и обновления классов сущностей в стиле DDD. В разделе 13.4 мы изменили классы сущностей для использования конструкторов или статических фабричных методов для создания и методов доступа для обновления. В этом разделе мы посмотрим, как можно было бы использовать эти новые подходы в приложении. Примеры взяты из версии приложения Book App (это приложение ASP.NET Core MVC) для третьей части. На рис. 13.6 показана страница, которую пользователь с правами администратора использует для добавления промоакции.

Затем мы реализуем код для добавления контроллера ASP.NET Core, чтобы отобразить страницу, показанную на рис. 13.6, и обновить сущность Book, когда пользователь ввел данные и щелкнул по кнопке «Обновить». Мы будем использовать два подхода: стандартный подход, описанный Эвансом, и библиотеку, предназначенную для работы напрямую с методами доступа DDD, – GenericServices. Следующий список позволяет сравнить оба подхода:

- вызов метода доступа AddPromotion через паттерн *Репозиторий*;

- вызов метода доступа `AddPromotion` с использованием вышеупомянутой библиотеки.

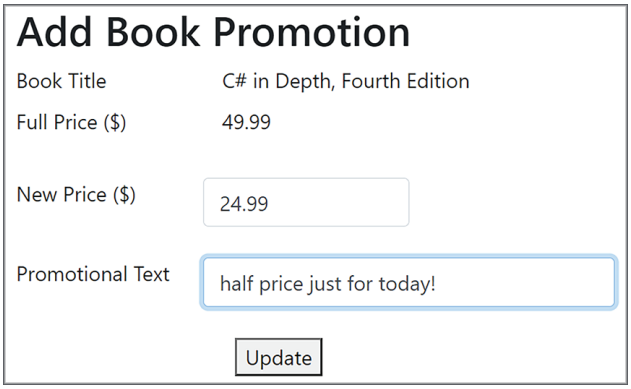


Рис. 13.6 Веб-страница, используемая для добавления промоакции для книги. В этом примере показаны название и полная стоимость книги, а также пользователю с правами администратора предлагается указать новую цену и добавить текст рядом с ней. При нажатии кнопки «Обновить» вызывается метод доступа `AddPromotion` с новыми данными, и если нет ошибок, вызывается метод `SaveChanges`, чтобы обновить книгу

Мы создадим код, который добавит новый отзыв в класс сущности `Book`. Обновления связей требуют, чтобы вы решили, как обрабатывать обновление. Мы реализуем пример с `AddReview` двумя способами, чтобы можно было сравнить оба подхода:

- добавление отзыва в класс сущности `Book` через паттерн *Репозиторий;*
- добавление отзыва в класс сущности `Book` с использованием библиотеки GenericServices.

13.5.1 Вызов метода доступа AddPromotion с помощью паттерна «Репозиторий»

В книге Эванса для обработки обращений к базе данных используется паттерн «Репозиторий». Определение Microsoft гласит: «Репозитории – это классы или компоненты, инкапсулирующие логику, необходимую для доступа к источникам данных. Они обобщают распространенные функции доступа к данным, обеспечивая лучшую сопровождаемость и отделяя инфраструктуру или технологии, используемые для доступа к базам данных, от уровня модели предметной области».

Есть много способов реализовать этот паттерн. Я решил использовать универсальный репозиторий, который будет работать с любой сущностью. В следующем листинге показан универсальный репозиторий, необходимый для примера с методом `AddPromotion`.

Листинг 13.7 Универсальный репозиторий, обрабатывающий основные команды базы данных

```
public class GenericRepository<TEntity>
    where TEntity : class
{
        protected readonly DbContext Context;

        public GenericRepository(DbContext context)
        {
            Context = context;
        }

        public IQueryable<TEntity> GetEntities()
        {
            return Context.Set<TEntity>();
        }

        public async Task<TEntity> FindEntityAsync(int id)
        {
            var entity = await Context.FindAsync<TEntity>(id);

            if (entity == null)
                throw new Exception("Could not find entity");

            return entity;
        }

        public Task PersistDataAsync()
        {
            return Context.SaveChangesAsync();
        }
}
```

Универсальный репозиторий будет работать с любым классом сущности

Репозиторию нужен DbContext базы данных

Этот метод находит и возвращает сущность с целочисленным первичным ключом

Возвращает запрос IQueryable типа сущности

Находит сущность по ее уникальному целочисленному первичному ключу

Элементарная проверка того, что сущность была найдена

Найденная сущность возвращается

Вызывает метод SaveChanges для обновления базы данных

Используя этот репозиторий, можно найти конкретную сущность `Book` и вызвать ее метод доступа `AddPromotion` с данными, предоставленными администратором. В следующем листинге показан код с использованием `GenericRepository<Book>`, который будет помещен в контроллер ASP.NET Core `AdminController`. У него есть два метода, оба с именем `AddPromotion`, но с разными параметрами и атрибутами. Первый метод `AddPromotion` вызывается для отображения страницы, показанной на рис. 13.6. Второй метод вызывается, когда пользователь нажимает кнопку «Обновить» и обрабатывает обновление сущности `Book` с рекламной акцией.

ПРИМЕЧАНИЕ Если вы незнакомы с ASP.NET Core, см. раздел 5.7, где приводится пошаговый обзор того, как работают контроллеры ASP.NET Core.

В следующем листинге показан `AdminController` с его конструктором и двумя методами. Обратите внимание, что комментарии есть только в новом коде, который использует репозиторий.

Листинг 13.8 Обновление метода AddPromotion с использованием паттерна «Репозиторий»

```
public class AdminController : Controller
{
    private readonly GenericRepository<Book> _repository;

    public AdminController(
        GenericRepository<Book> repository)
    {
        _repository = repository;
    }

    public async Task<IActionResult> AddPromotion(int id)
    {
        var book = await _repository.FindEntityAsync(id);

        var dto = new AddPromotionDto
        {
            BookId = id,
            Title = book.Title,
            OrgPrice = book.OrgPrice
        };

        return View(dto);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPromotion(AddPromotionDto dto)
    {
        if (!ModelState.IsValid)
        {
            return View(dto);
        }

        var book = await _repository
            .FindEntityAsync(dto.BookId);
        var status = book.AddPromotion(
            dto.ActualPrice, dto.PromotionalText);

        if (!status.HasErrors)
        {
            await _repository.PersistDataAsync();
            return View("BookUpdated", "Updated book…");
        }

        //Состояние ошибки;
        status.CopyErrorsToModelState(ModelState, dto);
        return View(dto);
    }
}
```

GenericRepository<Book> внедряется в контроллер

Копирует части Book, которые нужны, чтобы показать страницу

Использует репозиторий для чтения сущности Book

Вызывает метод доступа AddPromotion с двумя свойствами из dto

Метод доступа не вернул ошибок, поэтому мы сохраняем данные в базе данных

13.5.2 Вызов метода доступа AddPromotion с помощью библиотеки GenericServices

Хотя вызов методов доступа с использованием паттерна «Репозиторий» – рабочий вариант, в этом подходе есть дублирующийся код, например на первом этапе, когда мы копируем свойства в DTO/ViewModel (далее именуемый DTO) для отображения пользователю, и на втором этапе, когда возвращаемые данные в DTO превращаются в вызов метода доступа. Что бы произошло, если бы у нас был способ автоматизировать этот процесс?

В начале 2018 г., когда я закончил первое издание этой книги, я нашел способ автоматизировать обе части операций CRUD DDD и создал библиотеку с открытым исходным кодом `EfCore.GenericServices` (далее просто `GenericServices`). Эта библиотека автоматизирует большинство операций CRUD с обычными классами сущностей с изменяемыми свойствами и классами сущностей DDD с их конструкторами и методами доступа.

Одно из преимуществ использования данной библиотеки заключается в том, что она уменьшает объем кода, который вам нужно написать, по сравнению с подходом, где используется паттерн «Репозиторий». Она сэкономит вам около пяти строк кода в ASP.NET Core, и вам не нужно создавать репозиторий. Еще одно преимущество заключается в том, что код, который вы используете, одинаков для каждого обновления, отличаются только DTO. Библиотека позволяет копировать и вставлять код клиентской части, а затем изменять только тип DTO, чтобы переключиться на другой метод доступа, конструктор или статический фабричный метод.

> **ПРИМЕЧАНИЕ** Я разработал библиотеку `GenericServices` для работы с большинством, но не со всеми операциями CRUD. Она отлично справляется с простыми или средней сложности ситуациями, но все же не может охватить все случаи. Для более сложного варианта я пишу код вручную. Подробнее об этой библиотеке можно узнать на странице http://mng.bz/jBoP.

В оставшейся части этого раздела показано, как реализовать пример с методом `AddPromotion`, используя `GenericServices`. Сначала посмотрим на DTO на рис. 13.7, который определяет, какую сущность необходимо загрузить библиотеке, какие свойства загрузить для части чтения и какой метод доступа вызывать.

В следующем листинге показано использование библиотеки `GenericServices` вместо репозитория (листинг 13.8). Обратите внимание, что я оставил комментарии только для нового кода, который использует библиотеку.

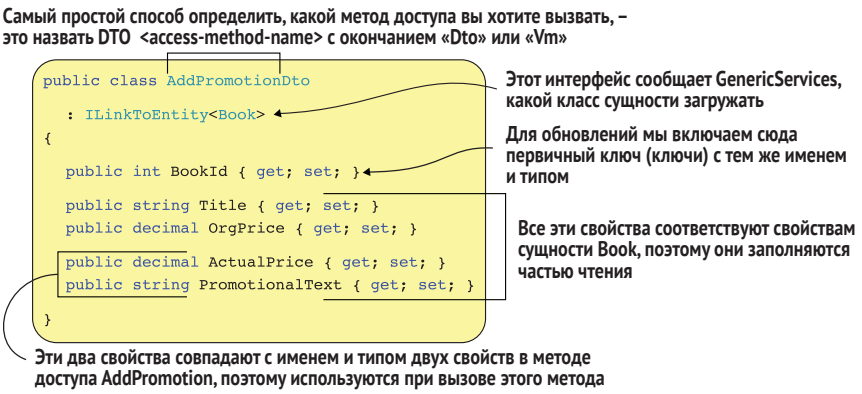


Рис. 13.7 DTO определяет, какой класс сущности читается и обновляется с помощью интерфейса `ILinkToEntity<T>`. При чтении он заполнит все свойства в DTO, у которых то же имя и тип, что и у связанного класса сущности – в данном случае сущности Book. Имя DTO используется, чтобы найти метод доступа для вызова, а свойства находятся путем сопоставления имен (используя стили написания PascalCase/camelCase) и их типов

Листинг 13.9 Обновление метода `AddPromotion` с помощью `GenericServices`

```
//public class AdminController : Controller
{
    private readonly ICrudServicesAsync _service;

    public AdminController(
        ICrudServicesAsync service)
    {
        _service = service;
    }

    public async Task<IActionResult> AddPromotion(int id)
    {
        var dto = await _service
            .ReadSingleAsync<AddPromotionDto>(id);

        return View(dto);
       }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPromotion(AddPromotionDto dto)
    {
        if (!ModelState.IsValid)
        {
            return View(dto);
        }

        await _service.UpdateAndSaveAsync(dto);
```

Сервис ICrudServicesAsync заимствован из GenericServices и внедряется через конструктор контроллера

ReadSingleAsync<T> читает в DTO, используя переданный первичный ключ

Метод UpdateAndSaveAsync вызывает метод доступа и, если ошибок не возникает, сохраняет метод доступа в базу данных

```
            if (!_service.HasErrors)
            {
                return View("BookUpdated", service.Message);
            }

            //Состояние ошибки;
            _service.CopyErrorsToModelState(ModelState, dto);
            return View(dto);
        }
    }
```

Как видите, кода намного меньше, и в каждом методе действия ASP.NET Core всего одна строка. Мой собственный анализ до и после показывает, что библиотека `GenericServices` сокращает время, необходимое для создания серверного приложения ASP.NET Core, на 10–20 %.

13.5.3 Добавление отзыва в класс сущности Book через паттерн «Репозиторий»

При обновлении навигационного свойства нужно выполнить еще один шаг: предварительно загрузить его. В листинге 13.4 методам доступа для добавления или удаления отзыва из сущности `Book` необходимо, чтобы вы заполнили резервное поле _reviews перед добавлением или удалением, поэтому нужно обновить репозиторий, который читает сущность `Book`, добавив коллекцию `Reviews`. Поскольку эта задача предназначена для сущности `Book`, мы создаем класс `BookRepository`, который унаследован от `GenericRepository`. Этот новый класс показан в следующем листинге.

Листинг 13.10 Добавление метода `LoadBookWithReviewsAsync` в репозиторий

Репозиторий наследуется от универсального репозитория для получения общих команд

```
public class BookRepository : GenericRepository<Book>
{
    public BookRepository(DbContext context)
        : base(context)
    { }

    public async Task<Book>
        LoadBookWithReviewsAsync(int bookId)
    {
        var book = await GetEntities()
            .Include(b => b.Reviews)
            .SingleOrDefaultAsync(
                b => b.BookId == bookId);
        if (book == null)
            throw new Exception("Could not find book");
```

GenericRepository нуждается в DbContext приложения

Загружает сущность Book с отзывами

Использует метод GetEntities из GenericRepository для получения запроса IQueryable<Book>

Убедитесь, что коллекция Review загружена вместе с книгой

Выбирает книгу с данным BookId

Элементарная проверка того, что сущность найдена

```
        return book;
    }
}
```

Возвращает книгу с загруженной коллекцией отзывов

Этот репозиторий заменяет `GenericRepository<Book>` из листинга 13.9. Данный фрагмент кода показывает, как вызвать метод `LoadBookWithReviewsAsync` в методе действия ASP.NET. Core, `POST`:

```
var book = await _repository
    .LoadBookWithReviewsAsync(dto.BookId);
book.AddReview(
    dto.NumStars, dto.Comment, dto.VoterName);
await _repository.PersistDataAsync();
```

13.5.4 Добавление отзыва в класс сущности Book с помощью библиотеки GenericServices

Для предварительной загрузки навигационных свойств библиотека `GenericServices` предоставляет атрибут `IncludeThen`, который вы добавляете в DTO. Этот атрибут позволяет определить имя навигационных свойств для `Include` или `ThenInclude`. В следующем листинге показан класс `AddReviewDto` с атрибутом `IncludeThen`.

Листинг 13.11 Класс `AddReviewDto` с атрибутом для загрузки отзывов

Атрибут IncludeThen включает в себя навигационное свойство Reviews

Имя DTO показывает, что он должен вызывать метод доступа AddReview

```
[IncludeThen(nameof(Book.Reviews))]
public class AddReviewDto
    : ILinkToEntity<Book>
{
    public int BookId { get; set; }

    public string Title { get; set; }

    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }
}
```

Сущность, с которой связан этот DTO, – это Book

Первичный ключ Book заполняется методом чтения и используется методом обновления

Заголовок считывается при чтении и используется, чтобы у пользователя было подтверждение, для какой книги он добавляет отзыв

Эти три свойства используются как параметры в методе доступа AddReview

После добавления атрибута `IncludeThen` любое чтение сущности будет включать навигационные свойства. Мы используем методы класса `GenericServices` `ReadSingleAsync<T>` и `UpdateAndSaveAsync(dto)` точно так же, как обращались бы к ним, когда нам не нужно было бы обновлять навигационные свойства.

13.6 Обратная сторона сущностей DDD: слишком много методов доступа

Мэтью Кригер прочитал одну из моих статей об использовании DDD с EF Core и оставил такой комментарий: «Вот чего я не могу понять: разве в конечном итоге у вас не будет слишком много методов доступа?» Он прав. В реальных приложениях можно получить много методов доступа. Когда вы создаете большое приложение, время, необходимое для написания методов доступа, быстро растет, особенно если нужно написать сотни таких методов.

Два моих клиента использовали DDD, и оба выбрали подход, позволяющий напрямую обновлять некоторые свойства без использования методов доступа. Один клиент хотел использовать JSON Patch для обновления сущностей, потому что это ускоряло создание страниц для клиентской части. Другой клиент использовал DDD, но обновлял некоторые свойства, позволяя AutoMapper «пробивать» закрытый метод записи и устанавливать значение. (Смотрите мое предупреждение в разделе 13.4.1.)

> **ОПРЕДЕЛЕНИЕ** JSON Patch – это способ отправки изменений в данные с помощью объекта JSON, соответствующий спецификации IETF RFC 6902. См. http://jsonpatch.com для получения дополнительной информации.

Оба клиента пытались ускорить разработку, и замечание «Это не правильный способ использования DDD» их не переубеждало. Мы договорились о таком подходе: если у свойства нет бизнес-правил (кроме атрибутов валидации), метод записи этого свойства можно сделать открытым. Я называю классы сущностей, использующие этот подход, *гибридными* сущностями DDD.

Вот пример. Если посмотреть на класс сущности `Book`, то видно, что у свойств `Title` и `Publisher` нет бизнес-логики, но они не должны быть пустыми, поэтому метод записи этих двух свойств можно сделать открытым, никак не влияя на бизнес-правила. Делая метод записи свойств открытым, вы избавляете себя от написания еще двух методов доступа, позволяя JSON Patch или AutoMapper обновлять эти свойства.

Некоторые сторонники DDD, возможно, раскритикуют этот гибридный вариант за несоблюдение паттерна, но я считаю это прагматическим решением.

> **ПРИМЕЧАНИЕ** Моя библиотека `GenericServices` может обнаруживать и использовать гибридные классы DDD. Если у такого

класса есть свойства с открытыми методами записи, она регистрирует класс сущности как гибридный. Гибридные классы позволяют `GenericServices` использовать JSON Patch или AutoMapper, чтобы настроить эти свойства напрямую без необходимости написания метода доступа. Смотрите мою статью на странице http://mng.bz/Wrj1.

13.7 Решение проблем с производительностью в DDD-сущностях

До сих пор мы рассматривали способы применения DDD к классам сущностей в EF Core. Но когда вы начинаете создавать настоящие приложения, иногда приходится оптимизировать производительность. Обычно проблемы с производительностью в приложении связаны с запросами, и DDD на них никак не влияет. Но если у вас проблемы с производительностью записи в базу данных, то может возникнуть необходимость обойти правила DDD. Вместо того чтобы отказываться от DDD, есть три способа продолжить использовать его с минимальным нарушением правил.

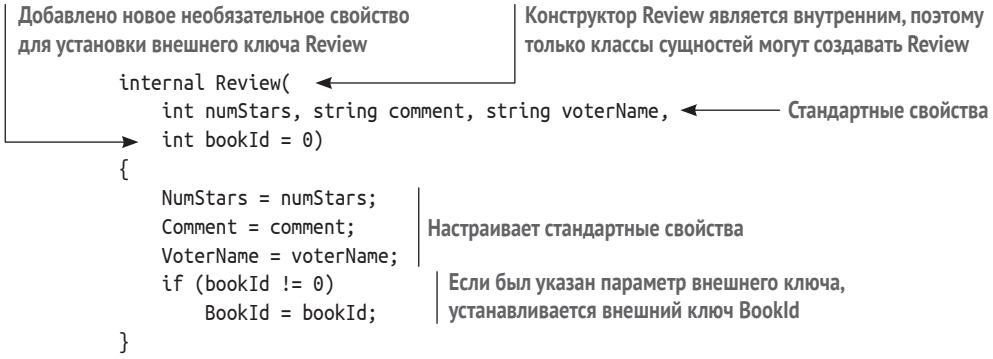
В качестве примера мы рассмотрим производительность добавления или удаления отзыва. Итак, мы загрузили все отзывы перед выполнением методов доступа для добавления и удаления. Если у вас всего несколько отзывов, нет проблем, но если ваш сайт похож на сайт Amazon, где у товаров могут быть тысячи отзывов, то загрузка всех их для добавления одного нового отзыва будет слишком медленной.

В разделе 3.4.3 я описываю способ добавления одного отзыва к книге путем создания этого отзыва и установки для внешнего ключа `BookId` первичного ключа `Book`. Такой подход означает, что вам не нужно загружать все отзывы, поэтому обновление будет быстрым. Но все решения из этого раздела нарушают правило DDD, согласно которому классы сущностей не должны ничего знать о коде базы данных. Поэтому в этом разделе мы рассмотрим три решения.

Каждое решение требует одного изменения: способа настройки внешнего ключа `BookId` в сущности `Review`. Это изменение немедленно нарушает правило, согласно которому сущности не должны знать о базе данных, но я не вижу другого способа обойти эту часть, хотя последний подход, который я описываю, близок к данному.

Начнем со следующего листинга, в котором показан обновленный конструктор `Review`. Обратите внимание, что у него есть модификатор доступа internal, а это значит, что его можно создать только в проекте Domain.Books. Использование модификатора доступа internal и необязательного параметра `BookId` в конструкторе станет понятнее по мере решения этой проблемы.

Листинг 13.12 Обновленный открытый конструктор Review с необязательным внешним ключом




```
internal Review(
    int numStars, string comment, string voterName,
    int bookId = 0)
{
    NumStars = numStars;
    Comment = comment;
    VoterName = voterName;
    if (bookId != 0)
        BookId = bookId;
}
```

АЛЬТЕРНАТИВНЫЙ ВАРИАНТ Есть и другой вариант – предоставить навигационное свойство, связывающее `Review` с сущностью `Book`. Этот вариант не позволяет сущности знать о внешних ключах, но нарушает правило DDD о минимизации связей. Выберите правило, которое вы хотите нарушить.

После изменения сущности `Review` можно использовать любой из трех вариантов:

- добавить код базы данных в свои классы сущностей;
- сделать конструктор Review открытым и написать код для добавления отзыва вне сущности;
- использовать события предметной области, чтобы попросить обработчика событий добавить отзыв в базу данных.

13.7.1 Добавить код базы данных в свои классы сущностей

Одно из решений состоит в том, чтобы у метода доступа `AddReview` был доступ к DbContext. Можно предоставить DbContext, добавив дополнительный параметр к методам доступа `AddReview/RemoveReview` или с помощью внедрения зависимости, как показано в разделе 6.1.10. В листинге 13.13 показаны два метода доступа для добавления и удаления отзыва. DbContext предоставляется в методах доступа через параметр.

ПРИМЕЧАНИЕ Я не мог использовать это решение в версии приложения Book App для третьей части, потому что чистая архитектура запрещает добавление каких-либо тяжеловесных библиотек, особенно связанных с базами данных, в проекты предметной области. Но использовал его в других приложениях.

Листинг 13.13 Предоставление DbContext приложения методам доступа

```
public void AddReview(
    int numStars, string comment, string voterName,
    DbContext context)
{
    if (BookId == default)
        throw new Exception("Book must be in db");

    if (context == null)
        throw new ArgumentNullException(
            nameof(context),
            "You must provide a context");

    var reviewToAdd = new Review(
        numStars, comment, voterName,
        BookId);

    context.Add(reviewToAdd);
}

public void RemoveReview (
    int reviewId,
    DbContext context)
{
    if (BookId == default)
        throw new Exception("Book must be in db");

    if (context == null)
        throw new ArgumentNullException(
            nameof(context),
            "You must provide a context");

    var reviewToDelete = context.Set<Review>()
        .SingleOrDefault(x => x.ReviewId == reviewId);

    if (reviewToDelete == null)
        throw new Exception("Not found");
    if (reviewToDelete.BookId != BookId)
        throw new Exception("Not linked to book");

    context.Remove(reviewToDelete);
}
```

... но добавляется новый параметр – DbContext

Метод доступа принимает обычные входные данные AddReview ...

Этот метод работает только с Book, которая уже есть в базе данных

Этот метод работает, только если предоставлен экземпляр DbContext

Создает отзыв и настраивает внешний ключ Review – BookId

Использует метод Add в DbContext, чтобы пометить новый отзыв для добавления в базу данных

Метод доступа RemoveReview принимает обычные входные данные – ReviewId

Этот метод работает только с Book, которая уже есть в базе данных

Этот метод работает, только если предоставлен экземпляр DbContext

Читает отзыв на удаление

Элементарная проверка того, что сущность найдена

Если нет связи с этой сущностью Book, выбрасывается исключение

Удаляет отзыв

Данное решение нарушает следующие правила DDD:

- методы доступа для добавления и удаления отзывов содержат функции базы данных;
- сущность `Review` знает о внешнем ключе `BookId`.

ПРИМЕЧАНИЕ Библиотека `GenericServices` поддерживает внедрение DbContext через параметр. Когда она вызывает конструкторы DDD, статические фабричные методы или методы доступа, она ищет параметры типа `DbContext` или типа DBContext приложения и заполняет их контекстом DBContext, в котором была зарегистрирована.

13.7.2 Сделать конструктор Review открытым и написать код для добавления отзыва вне сущности

Данное решение удаляет функции базы данных, представленные в разделе 13.7.1, из методов доступа `Book` и помещает их в другой проект (скорее всего, BizLogic). Это решение делает сущность `Book` чище, но требует, чтобы модификатор доступа конструктора `Review` был изменен на `public`. Есть и обратная сторона: любой может создать экземпляр сущности `Review`.

Код для добавления и удаления отзыва такой же, как и в листинге 13.4, но теперь он находится в собственном классе. Это решение нарушает следующие правила DDD:

- сущность `Book` не отвечает за связанные с ней сущности `Review`;
- у `Review` есть открытый конструктор, поэтому любой разработчик может создать отзыв;
- сущность `Review` знает о внешнем ключе `BookId`.

13.7.3 Использовать события предметной области, чтобы попросить обработчик событий добавить отзыв в базу данных

Последнее решение – использовать событие предметной области (см. главу 12) для отправки запроса обработчикам событий, которые добавляют или удаляют отзыв. На рис. 13.8 показан метод доступа `AddReviewViaEvents` в сущности `Book` слева, а справа метод `SaveChanges` (или `SaveChangesAsync`) запускает `AddReviewHandler`.

На рис. 13.8 показан только пример с `AddReview`, но `RemoveReview` отправит идентификатор `ReviewId` в `RemoveReviewHandler`, задача которого – найти и удалить этот отзыв. Этот подход меньше всех отклоняется от DDD, потому что сущность `Book` по-прежнему отвечает за управление отзывами, связанными с ней. Кроме того, сущность `Review` может сохранить свой модификатор доступа internal, чтобы никакой код вне проекта классов сущностей не мог создать отзыв. Но тем не менее у него все же есть обратная сторона, которая есть у всех решений: сущность `Review` знает о внешнем ключе базы данных.

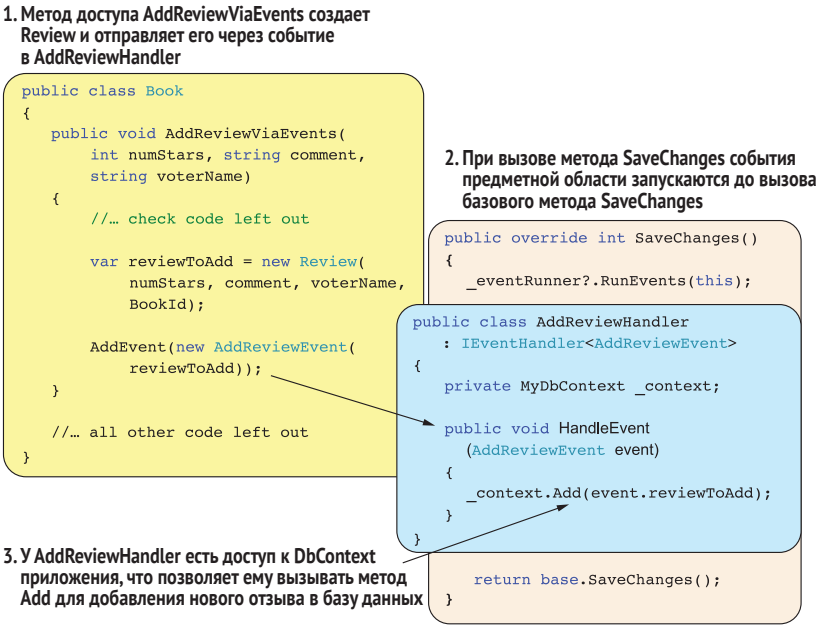


Рис. 13.8 Решение, использующее события для добавления одного отзыва без загрузки всех отзывов в сущность Book. У сущности Book есть метод доступа `AddReviewsViaEvents`, который создает отзыв и отправляет его в событии предметной области в обработчик событий. Когда вызывается событийно-расширенный метод `SaveChanges` или `SaveChangesAsync`, он находит и запускает `AddReviewHandler`, предоставляя событие предметной области в качестве параметра. Обработчик событий может получить доступ к DbContext приложения, поэтому может вызвать метод `Add`, чтобы добавить новый отзыв в базу данных. Затем базовый метод `SaveChanges` или `SaveChangesAsync` обновляет базу данных, внося изменения

13.8 Три архитектурных подхода: сработали ли они?

Опыт создания и улучшения приложения Book App стал отличным испытанием для трех архитектурных подходов во время разработки. В начале у этого приложения было 9 проектов, а к концу главы 16 – уже 23 – серьезное изменение с большим количеством рефакторинга для поддержки новых функций. В этом разделе я резюмирую свой опыт использования этих подходов в приложении Book App с начала третьей части до конца главы 16.

13.8.1 Модульный монолит, реализующий принцип разделения ответственностей с помощью проектов

Я знал об этом подходе, но прежде не использовал его в приложении. По моему опыту, он хорошо сработал; на самом деле все было намно-

го лучше, чем я думал. Я бы снова использовал его для любого среднего и большого приложения. Следовать этому подходу означало, что каждый проект был небольшим и целенаправленным, и присвоение проекту названия, говорящего о том, что он делает, упростило навигацию по коду.

Пользуясь многоуровневой архитектурой (см. раздел 5.2) какое-то время, я знаю, что сервисный уровень может стать действительно большим и сложным для понимания (иногда его называют *большим комком грязи*). Я попытался сгладить эту проблему, сгруппировав связанный код в папки, но я не могу быть полностью уверен, связан ли код из папки A с кодом в других папках. Когда я использую многоуровневую архитектуру, если я тороплюсь, то обычно пишу что-то новое, вместо того чтобы проводить рефакторинг старого кода. Я не могу найти время, чтобы понять, используется ли код где-то еще или использует ли он что-то, о чем я не знаю.

В отличие от этого, модульный монолит предоставляет небольшие целенаправленные проекты. Я знаю, что весь код в проекте выполняет одну задачу и проекты ссылаются только на имеющие к ним отношение другие проекты. Такой подход упрощает понимание кода, и я гораздо более склонен к рефакторингу старого кода, так как у меня меньше шансов испортить что-то еще.

Я обнаружил одну вещь – я ссылаюсь на проект Book Display, который содержал исходный код приложения Book App из первой части. Этот уровень содержит несколько полезных классов и перечисления, которые можно было бы использовать в других таких проектах. Я нарушал правила модульного монолита, ссылаясь на проект, в котором было много кода, не имеющего отношения к связанному проекту. Мне следовало вынести эти общие классы в отдельный проект, но я спешил закончить книгу, а на первый проект Book Display было легко ссылаться (как на настоящей работе!). Модульный монолит помогает разделить код, но требует от разработчика соблюдения правил.

ПРИМЕЧАНИЕ Мне пришлось вернуться к приложению Book App в главе 16, чтобы добавить новые версии некоторых функций отображения, поэтому я воспользовался возможностью создать проект `BookApp.ServiceLayer.DisplayCommon.Books`, содержащий весь общий код. Этот проект удаляет связь между функциями запросов и делает код намного проще для понимания и рефакторинга.

Вот несколько советов по использованию модульного монолита:

- используйте для своих проектов правило иерархического именования. Такие имена, как `BookApp.Persistence.EfCoreSql.Books`, например, упрощают поиск;
- не завершайте название проекта именем класса. Используйте что-то вроде … `Books`, а не… `Book`. Я назвал несколько проектов… `Book`, в результате чего мне требовалось добавлять к каждому

классу `Book` его полное пространство имен – в данном случае `BookApp.Domain.Book.Book`;
- вы будете ошибаться в названиях проектов. Я назвал один проект `BookApp.Infrastructure.Books.EventHandlers`, но по мере роста приложения Book App и расширения проекта мне пришлось изменить его на `BookApp.Infrastructure.Books.CachedValues`;
- если вы изменяете имя проекта в Visual Studio, выбрав проект и введя новое имя, то вы не меняете имя папки. Я обнаружил, что такая ситуация сбивает с толку в GitHub, поэтому переименовывал и папку, что означало редактирование файла решения (есть хороший инструмент, который может сделать это за вас: https://github.com/ModernRonin/ProjectRenamer);
- вам понадобится Visual Studio 16.8.0 или более поздняя версия, если у вас будет много проектов в приложении, потому что Visual Studio 16.8 намного быстрее, чем более старые версии, справляется с большим количеством проектов в решении (VS Code всегда работал быстро со множеством проектов).

13.8.2 Принципы DDD как в архитектуре, так и в классах сущностей

Я знаком с использованием DDD, и, как и ожидал, оно сработало. Вот список возможностей, которые упростили разработку приложения Book App:

- каждый класс сущности содержал весь код, необходимый для создания или обновления этой сущности и всех агрегатов. Если мне нужно было что-то изменить, я знал, где искать, и знал, что другой версии этого кода больше нигде не было;
- методы доступа были особенно полезны, когда я использовал события предметной области в главе 15;
- методы доступа оказались даже более полезными, когда я использовал события интеграции в главе 16, потому что мне нужно было перехватывать все возможные изменения в сущности `Book` и ее агрегатах, что было легко сделать, добавляя события интеграции в каждый метод доступа и статический фабричный метод в сущности `Book`. Если бы я не мог перехватывать все изменения таким образом, мне пришлось бы обнаруживать изменения, используя свойство сущности `State`, а я знаю по опыту, что обнаружение изменений сложно реализовать;
- ограниченный контекст, который допускал два разных контекста EF, `BookDbContext` и `OrderDbContext`, также сработал. Миграция двух частей одной той же базы данных (см. раздел 9.4.5) прошла прекрасно.

13.8.3 Чистая архитектура согласно описанию Роберта С. Мартина

Я не впервые использовал чистую архитектуру, так как работал над клиентским приложением, где использовался этот подход, но здесь я впервые начал с нуля. Я был гораздо лучше осведомлен о том, где следует разместить различные части приложения. В целом я обнаружил, что уровни чистой архитектуры полезны, но мне пришлось изменить одну вещь, которую я опишу в конце этого раздела.

К концу главы 16 книжное приложение Book App состояло из пяти уровней:

- *предметная область* – уровень, содержащий классы сущностей;
- *хранение* – уровень, содержащий DbContext и другие классы базы данных;
- *инфраструктура* – уровень, собравший несколько проектов, таких как заполнение базы данных и обработчики событий;
- *сервисный уровень* – содержащий код для адаптации нижних уровней к клиентской части;
- *пользовательский интерфейс* – уровень, который содержит приложение ASP.NET Core.

На рис. 13.9 показаны все пять уровней с количеством проектов на каждом уровне после написания всех глав.

Рис. 13.9 Пять уровней приложения Book App в третьей части с количеством проектов на каждом уровне после завершения главы 16. Смотрите ветку `Part3` связанного репозитория GitHub для каждого проекта

Основная проблема заключалась в том, чтобы встроить DbContext в чистую архитектуру. Чистая архитектура гласит, что база данных должна находиться на внешнем кольце с интерфейсами для доступа.

Проблема состоит в том, что нет простого интерфейса, который можно использовать для DbContext. Даже если бы я использовал паттерн «Репозиторий» (а я этого не делал), проблема бы осталась, потому что DbContext приложения должен быть определен глубоко в уровнях.

Одно из правил чистой архитектуры, которое мне не понравилось, но которого я придерживался, заключается в том, что на уровне предметной области не должно добавляться тяжеловесных внешних зависимостей (например, библиотек NuGet). Это правило требовало, чтобы я проделал больше работы. Например, в главе 15 у меня был код, который помечал каждую сущность `Book`, когда она добавлялась или обновлялась. Проще было бы передать класс EF Core `EntityEntry` в метод `LogAddUpdate` на уровне предметной области. Кроме того, в главе 16 я хотел использовать собственный тип с Cosmos DB, и мне пришлось применять команды конфигурации Fluent API, чтобы настроить его. Я бы предпочел добавить к классу атрибут `[Owned]`, который избавил бы меня от добавления метода `OnModelCreating` в класс `CosmosDbContext`, чтобы добавить дополнительные команды конфигурации Fluent API. В следующий раз я думаю добавить базовый пакет NuGet для обработки этих функций.

Резюме

- Архитектура, которая используется для создания приложения, должна помочь вам сосредоточиться на функции, которую вы добавляете, при этом хорошо разделяя код, чтобы было проще проводить рефакторинг.
- Предметно-ориентированное проектирование (DDD) дает множество хороших рекомендаций по созданию приложения, но в этой главе основное внимание уделяется классам сущностей EF Core и DbContext приложения.
- DDD-сущности управляют тем, как они создаются и обновляются; сущность должна гарантировать, что данные внутри нее являются допустимыми.
- В DDD есть множество правил, гарантирующих, что разработчики могут приложить все свои усилия для удовлетворения потребностей предметной области (бизнес-потребностей), которые им было предложено реализовать.
- DDD группирует сущности в агрегаты и гласит, что одна сущность в группе, известная как корневой объект, управляет данными и связями в агрегате.
- Ограниченный контекст – это основная концепция DDD. В этой главе рассматривается только то, как ограниченный контекст можно применить к DbContext приложения.
- Чтобы обновить DDD-сущность, вызывается метод в классе сущности. В этой книге они называются методами доступа.

- Чтобы создать новый экземпляр DDD-сущности, используется конструктор с определенными параметрами или статический фабричный метод создания, возвращающий статус с результатом создания.
- Чтобы обновить DDD-сущность, для начала загрузите ее, чтобы можно было вызвать метод доступа. Это можно сделать с помощью обычного кода EF Core, репозитория или библиотеки `EFCore.GenericServices`.
- Библиотека `EFCore.GenericServices` экономит время на разработку. Она устраняет необходимость написания репозитория и может находить и вызывать методы доступа, используя имя и свойства в DTO.
- Обновление связей коллекции может быть медленным, если у вас много существующих записей в коллекции. В этих случаях есть три способа улучшить производительность.
- Обзор применения трех архитектурных подходов в главе 16 показывает, что все они улучшили приложение Book App и упростили его рефакторинг. Все подходы сработали, но хотелось бы выделить модульный монолит и DDD.

Для читателей, знакомых с EF6.x:

- в EF6.x нельзя полностью создавать DDD-сущности, потому что нельзя сделать навигационные свойства коллекции свойствами с доступом только на чтение. EF Core решил эту проблему, используя резервные поля.

14 Настройка производительности в EF Core

В этой главе рассматриваются следующие темы:

- решение относительно того, какие проблемы с производительностью нужно устранять;
- использование методик для обнаружения проблем с производительностью;
- использование паттернов, способствующих хорошей производительности;
- поиск паттернов, вызывающих проблемы с производительностью.

Это первая из трех глав, посвященных настройке производительности доступа к базе данных. Обсуждение того, что нужно улучшить, а также где и как улучшить код доступа к базе данных, разделено на три части:

- *часть 1* – понимание производительности, разница между скоростью и масштабируемостью, принятие решения относительно того, что нужно настроить для повышения производительности, и определение затрат на это;
- *часть 2* – методики, которые можно использовать для поиска проблем с производительностью, и использование журналирования EF Core для помощи в этом;

- *часть 3* – целый ряд паттернов доступа к базе данных, хороших и плохих, которые помогут диагностировать и устранить многие проблемы с производительностью в EF Core.

В главе 15 мы применим подходы, показанные в этой главе, к запросу списка книг приложения Book App. Мы начнем с настройки кода EF Core, а затем перейдем к более сложным методам, таким как добавление SQL-запросов для достижения максимальной производительности доступа к базе данных.

14.1 Часть 1: решаем, какие проблемы с производительностью нужно исправлять

Прежде чем описывать, как находить и исправлять проблемы с производительностью, хочу сделать обзор того, что такое производительность. Хотя вы и можете игнорировать ее в начале проекта, некоторые концепции могут помочь вам позже, когда кто-то скажет: «Приложение слишком медленное; исправьте это».

Когда речь идет о *производительности* приложения, обычно подразумевается то, насколько быстро приложение обрабатывает запросы: например, сколько времени требуется API, чтобы вернуть ответ на конкретный запрос, или сколько пользователь должен ждать при поиске конкретной книги. Я называю эту часть производительности приложения *скоростью* и использую термины *быстро* и *медленно*, чтобы описать ее.

Другой аспект – это то, что происходит со скоростью приложения, когда у него много одновременных запросов. Быстрый сайт с несколькими пользователями может стать медленным, если увеличится количество одновременных пользователей. Эта ситуация называется *масштабируемостью* приложения – способность работать быстро, даже если к нему начинает обращаться большое количество пользователей. Масштабируемость часто измеряется *пропускной способностью* – количеством запросов, которые приложение может обработать за одну секунду.

14.1.1 Фраза «Не занимайтесь настройкой производительности на ранних этапах» не означает, что нужно перестать думать об этом

Практически все говорят, что не стоит задумываться о производительности на ранних этапах; цель номер один – сначала заставить приложение работать должным образом. Кенту Беку приписывают изречение: «Сделайте так, чтобы это работало. Сделайте, чтобы это работало правильно. Сделайте, чтобы это работало быстро», – в котором последовательно рассматриваются этапы создания приложения

и настройка производительности идет в последнюю очередь. Я полностью согласен с этим, но есть три оговорки:

- убедитесь, что любые паттерны проектирования, которые вы используете, сами не имеют проблем с производительностью. В противном случае вы с первого дня будете усугублять проблемы. См. раздел 14.4;
- не пишите код, затрудняющий поиск и устранение проблем с производительностью. Если вы смешаете код доступа к базе данных с другим кодом, например с кодом клиентской части, то изменения производительности могут стать запутанными и их трудно будет тестировать. См. раздел 14.4.6;
- выбирайте правильную архитектуру. В настоящее время масштабируемость веб-приложений легче улучшить, запустив несколько экземпляров приложения. Но если у вас есть приложение, требующее высокой масштабируемости, то вам может помочь архитектура CQRS. Об этом рассказывается в главе 16.

Часто бывает трудно предсказать, с какими проблемами производительности вы столкнетесь, поэтому разумно будет подождать, пока ваше приложение не начнет обретать форму. Но, немного подумав заранее, вы можете избавить себя от лишней головной боли в дальнейшем, когда обнаружите, что ваше приложение слишком медленное.

14.1.2 Как определить, что работает медленно и требует настройки производительности?

Проблема с такими терминами, как «быстро», «медленно» и «высокая нагрузка», заключается в том, что они субъективны. Вы можете считать, что ваше приложение работает быстро, а ваши пользователи могут считать его медленным. Если придерживаться субъективных взглядов на производительность приложения, то это не поможет, поэтому ключевые вопросы заключаются в следующем: имеет ли значение в данном случае скорость и какой она должна быть?

Вы должны помнить, что в приложениях, ориентированных на людей, важна общая скорость, но не менее важны и *ожидания пользователя* относительно того, насколько быстрой должна быть определенная функция. Например, Google показал, насколько быстрым может быть поиск; поэтому мы ожидаем, что любой поиск будет быстрым. И наоборот, при оплате покупки в интернете необходимо указать свой адрес, номер кредитной карты и т. д. – мы не ожидаем, что процесс будет быстрым (хотя если все будет слишком медленно или сложно, мы откажемся от этого!).

Когда вы думаете о том, где нужно улучшить производительность, необходимо быть избирательным; в противном случае придется приложить много усилий, а эффект будет небольшим. Однажды я разработал небольшой сайт для онлайн-торговли, на котором было чуть более 100 различных запросов и обновлений для 20 таблиц базы данных. Бо-

лее 60 % обращений к базе данных были со стороны администратора, а некоторые из них использовались редко. Возможно, 10 % обращений к базе данных касались пользователей, оплачивающих покупки. Этот анализ помог мне решить, где нужно приложить усилия.

На рис. 14.1 показано, что происходит, когда вы применяете тот же анализ ожиданий пользователя относительно скорости доступа к базе данных приложения Book App. Этот анализ охватывает вывод списка книг и их поиск, размещение заказа и несколько команд администратора, начиная от обновления даты публикации книги (быстро) до очистки и повторного ввода данных по всем книгам (довольно медленно).

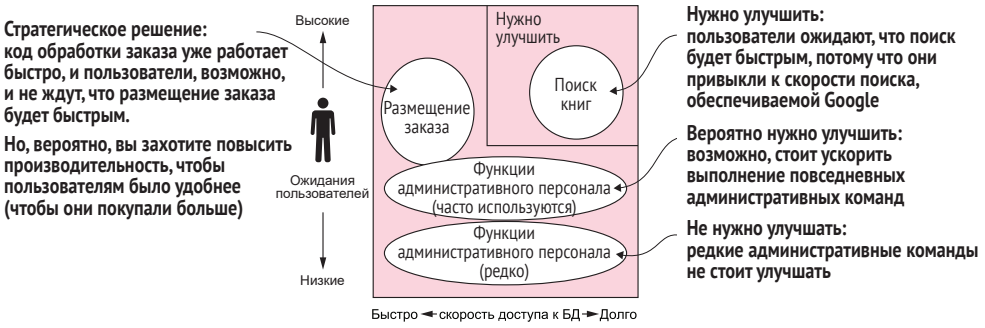


Рис. 14.1 Различные возможности приложения Book App, оцененные в соответствии с ожиданиями пользователя относительно скорости доступа и фактической сложности и скорости доступа к базе данных. Тип пользователя и его ожидания оказывают большое влияние на то, где требуется улучшение производительности

После того как вы проанализируете свое приложение, вы должны получить список функциональности, который заслуживает улучшения производительности. Но, прежде чем начать, нужны четкие показатели:

- *определите функциональность* – какой именно запрос или команду нужно улучшить, и при каких обстоятельствах она работает медленно (например, количество одновременных пользователей)?
- *получите хронометраж* – сколько времени занимает эта функциональность сейчас и насколько быстрым она должна быть?
- *оцените затраты на исправление* – во сколько обойдется улучшение? Когда следует остановиться?
- *докажите, что все по-прежнему работает* – у вас есть способ подтвердить, что функциональность работает правильно, прежде чем приступить к улучшению производительности, и что она все еще работает после изменения производительности?

СОВЕТ На странице http://mng.bz/G62D можно найти старую, но по-прежнему полезную статью об общей настройке производительности.

14.1.3 Затраты на поиск и устранение проблем с производительностью

Прежде чем углубиться в поиск и устранение проблем с производительностью, хочу отметить, что оптимизация производительности приложения требует определенных затрат. Требуется время на разработку и усилия по поиску, улучшению и повторному тестированию производительности приложения. Как показано на рис. 14.1, нужно тщательно выбирать то, что вы планируете улучшить.

Много лет назад я написал статью, в которой измерял прирост производительности при доступе к базе данных EF6.x по сравнению со временем, которое мне потребовалось, чтобы добиться этого улучшения. Результаты этой работы показаны на рис. 14.2. Я начал с существующего запроса EF6.x (1 на горизонтальной шкале), а затем применил два шага (2 и 3) для улучшения, все еще используя EF6.x. В конце я оценил время, которое потребуется для написания версии, использующей только SQL-запросы (4 на горизонтальной шкале).

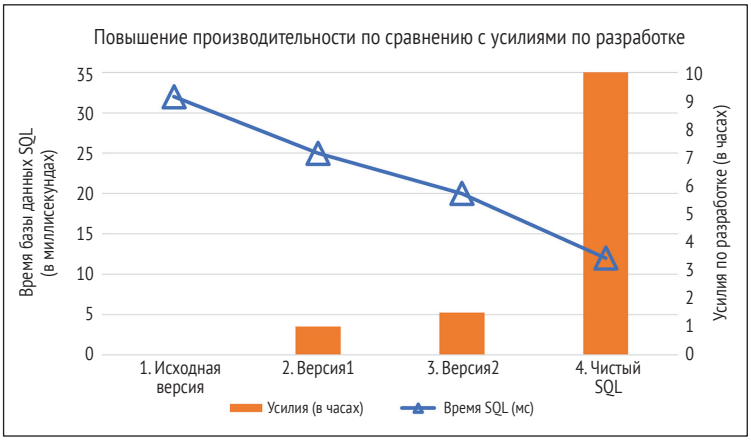


Рис. 14.2 Компромисс между производительностью базы данных и усилиями по разработке улучшения доступа к базе данных EF для трех этапов. Время разработки отображается в виде гистограммы (часы: левая шкала), а скорость доступа к базе данных отображается в виде линии (миллисекунды: правая шкала). Обратите внимание на почти экспоненциальный рост усилий по разработке по сравнению с почти линейным сокращением времени доступа к базе данных

Смысл рис. 14.2 состоит в том, чтобы показать, что добиться значительного улучшения производительности непросто. У меня был экспоненциальный рост усилий по разработке в сравнении с почти линейным сокращением времени доступа к базе данных. Поэтому стоит рассматривать проблему в целом. Хотя запросы к базе данных могут быть медленными, решение может заключаться в изменении других частей приложения. В случае с веб- и мобильными приложениями есть еще несколько возможностей:

- *HTTP-кеширование* – кеширование позволяет запомнить результат запроса в памяти и вернуть копию, если запрос повторится, что избавляет от необходимости обращения к базе данных. Добавление кеширования требует работы, но может сильно повлиять на воспринимаемую производительность;
- *вертикальное и горизонтальное масштабирование* – облачный хостинг позволяет платить за более мощные хост-компьютеры (известное как вертикальное масштабирование в Azure) и/или запуск дополнительных экземпляров веб-приложения (известное как горизонтальное масштабирование в Azure). Такой подход может быстро решить множество небольших проблем, связанных с производительностью, особенно если проблема заключается в масштабируемости.

Я не призываю к небрежному программированию. Конечно, в этой книге я стараюсь показать хорошие техники. Но, выбрав EF Core вместо написания чистых команд SQL, вы уже выбрали меньшее время разработки с (возможно) более медленным доступом к базе данных. В конце концов, речь всегда идет о соотношении результатов и трудозатрат, поэтому следует оптимизировать производительность только тех частей приложения, которым требуется дополнительная скорость или масштабируемость.

14.2 Часть 2: методы диагностики проблем с производительностью

В первой части вы определили, какие части приложения нуждаются в улучшении и каков объем требуемых улучшений. Следующий шаг – найти код, связанный с медленной функциональностью, и диагностировать проблему.

Эта книга посвящена EF Core, поэтому мы сконцентрируемся на обращениях к базе данных, но такие обращения редко существуют сами по себе. Необходимо изучить приложение, чтобы найти код доступа к базе данных, который влияет на производительность. На рис. 14.3 показан трехэтапный подход, который я использую для выявления узких мест в производительности. Эти этапы будут подробно рассмотрены в следующих трех подразделах.

ПРЕДУПРЕЖДЕНИЕ Измерение количества времени, необходимого ASP.NET Core для выполнения команды в режиме отладки, может давать неверные цифры, потому что могут быть использованы медленные методы журналирования. Они могут значительно увеличить время выполнения каждого HTTP-запроса. Я рекомендую тестировать программное обеспечение, собранное в режиме релиза, чтобы получить более репрезентативные цифры.

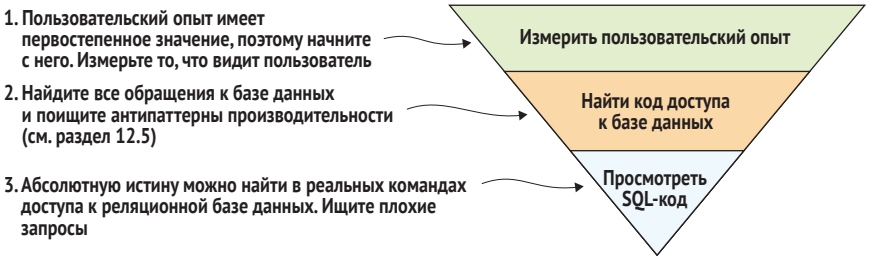


Рис. 14.3 Для обнаружения проблем с производительностью базы данных необходимо начать с того, что видит пользователь, а затем перейти к коду. Найдя код доступа к базе данных, вы проверяете, используются ли в нем оптимальные стратегии, описанные в этой главе. Если этот шаг не улучшит ситуацию, нужно просмотреть команды SQL, отправляемые в базу данных, и подумать о способах их улучшения

14.2.1 Этап 1. Получить хорошее общее представление, оценив опыт пользователей

Прежде чем заняться поисками проблем с производительностью, нужно подумать о пользовательском опыте, потому что он имеет значение. Можно повысить скорость доступа к базе данных на 500 %, но если работа с базой данных составляет лишь небольшую часть общей картины, такое улучшение не сильно поможет.

Во-первых, нужно найти инструмент, который измерит, сколько времени занимает конкретный запрос или функциональность. Выбор инструмента будет зависеть от типа приложения. Вот список бесплатных инструментов, позволяющих узнать общее время выполнения запроса:

- для приложений Windows можно использовать профилировщик производительности в Visual Studio;
- для сайтов можно использовать браузер в режиме разработчика, чтобы получить хронометраж (я использую Google Chrome);
- для ASP.NET Core Web API можно использовать Azure Application Insights локально в режиме отладки;
- и не забывайте про журналирование: журналы ASP.NET Core и EF Core включают хронометраж.

ПРИМЕЧАНИЕ Все перечисленные мною инструменты бесплатны, но для тестирования и профилирования всевозможных систем также доступно множество коммерческих (платных) инструментов.

На рис. 14.4 показана временная шкала для приложения Book App до настройки производительности, измеренная браузером Google Chrome в режиме разработчика (F12), но большинство браузеров содержат те же функции. На рисунке показан только один хронометраж,

но для каждого запроса нужно проводить несколько измерений, поскольку они могут отличаться. Кроме того, чтобы получить понимание того, где существуют проблемы с производительностью, нужно попробовать разные комбинации сортировки и фильтрации при выводе списка книг. Смотрите главу 15, где приводится пример хронометража для нескольких таких комбинаций.

> **ПРИМЕЧАНИЕ** Приложение Book App в ветке `Part3` перехватывает запись в журнал ASP.NET Core, `RequestFinished`, содержащую общее время выполнения HTTP-запроса. Если повторить тот же запрос, эта функция предоставит максимальный, минимальный и средний хронометраж. Доступ к этой функции можно получить через команду меню **Admin** > **Timings last URL**.

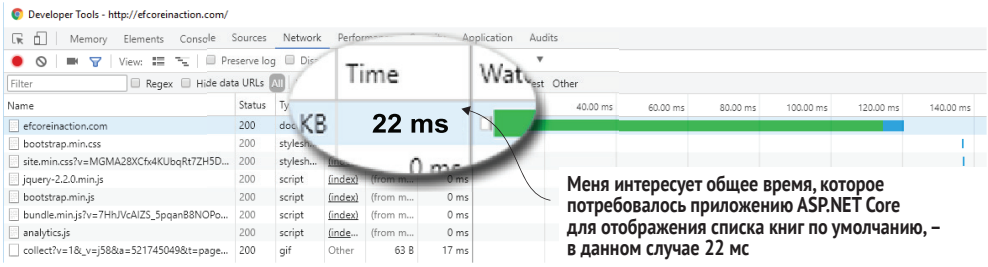


Рис. 14.4 Использование браузера Google Chrome в режиме разработчика, чтобы узнать, сколько времени требуется приложению Book App для отображения 700 книг при использовании EF Core 5, прежде чем мы приступим к настройке производительности. Эта функция уже работает быстро, но в главе 15, когда мы доведем отображение до 100 000 книг, начнутся проблемы

14.2.2 Этап 2. Найти весь код доступа к базе данных, связанный с оптимизируемой функцией

После выбора части приложения, производительность которой вы хотите улучшить, необходимо найти весь код доступа к базе данных, задействованный в этой функциональности. После того как вы нашли код доступа к базе данных, просмотрите его в поисках антипаттернов производительности (см. разделы 14.5 и 14.6). Это позволяет быстро найти и устранить проблемы. Это не самый надежный способ, но через некоторое время вы сможете чувствовать, что может быть причиной проблем.

Например, если посмотреть на список книг в приложении Book App, то наиболее очевидным проблемным местом является подсчет средней оценки. Средняя оценка используется не только для того, чтобы показать ее пользователю, но и для сортировки и фильтрации отображаемых книг. Выполнение различных тестов показало, что сортировка или фильтрация по средней оценке была медленной, но

я заметил эту проблему, только когда посмотрел на вывод журнала EF Core (раздел 14.2.3).

В приложении Book App не так много операций записи, только добавление отзывов и добавление или удаление промоакций, и они работают быстро, но во многих приложениях операции записи могут быть узким местом. Проблемы с производительностью при записи, возможно, сложнее диагностировать, поскольку необходимо учитывать две части: время, необходимое EF Core для обнаружения и связывания изменений с данными, и время, необходимое для записи в базу данных. Для операций записи важен общий хронометраж, поскольку он включает обе части (см. раздел 14.6).

14.2.3 Этап 3. Проверить SQL-код, чтобы выявить низкую производительность

Конечным источником производительности запросов к базе данных является SQL-код. В журналах EF Core будет записан SQL-код, отправленный в базу данных, а также время, затраченное на запрос. Я расскажу, как использовать эту информацию для поиска проблем с производительностью, но для начала позвольте мне описать, как получить доступ к сообщениям журналов, создаваемых EF Core. Вот как выглядят этапы их получения:

1. Описание сообщений журналов, создаваемых EF Core.
2. Получение информации из журналов.
3. Извлечение команд SQL, отправленных в базу данных.

ОПИСАНИЕ СООБЩЕНИЙ ЖУРНАЛОВ, СОЗДАВАЕМЫХ EF CORE

.NET Core определяет стандартный интерфейс журналирования, который может использоваться из любого кода. EF Core производит значительное количество сообщений, которые обычно собираются приложением, в котором он выполняется. Информация в журнале подразделяется на `LogLevel`, который варьируется от самой подробной информации на уровне `Trace` (0) до `Critical` (5). В производственном окружении можно ограничить вывод уровнем `Warning` (3) и выше, но при работе в режиме отладки желательно использовать уровень `Information`, поскольку начиная с этого уровня сообщения от EF Core (и ASP.NET Core) содержат полезную информацию и хронометраж.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ ИЗ ЖУРНАЛОВ

Один из способов доступа к журналам – использование так называемого *поставщика журналов*. Журналирование настолько полезно, что большинство приложений включают в себя код для настройки этих поставщиков. В приложении ASP.NET Core, например, поставщик (поставщики) журналов настраивается во время запуска (http://mng.bz/KH6W), поэтому вы можете получить журналы из приложения, работающего в режиме отладки, либо из реального приложения.

Еще один способ сбора сведений из журналов – использование `LogTo` в EF Core 5 внутри модульных тестов. Эта функция обеспечивает простой способ получения информации из журнала EF Core. В листинге 14.1 показан один из способов ее использования, но я рекомендую прочитать главу 17, посвященную модульному тестированию кода EF Core.

> **ПРИМЕЧАНИЕ** Если вы используете библиотеку модульных тестов xUnit (https://xunit.net), нельзя выводить данные с помощью метода `Console.Writeline`, поскольку xUnit запускает тесты параллельно, и у вас получится список несвязанных записей. Эта тема подробно рассматривается в разделе 17.11.1, а также там рассказывается о том, как выполнять вывод в консоль из xUnit.

Листинг 14.1 Получение сообщений журналов EF Core в модульном тесте

Хранит все сообщения журналов, которые выводит EF Core

DbContextOptionsBuilder<T> – это способ создания настроек, необходимых для создания контекста

Указывает, что используется база данных SQL Server, и принимает строку подключения

```
var logs = new List<string>();
var builder =
    new DbContextOptionsBuilder<BookDbContext>()
    .UseSqlServer(connectionString)
    .EnableSensitiveDataLogging()
    .LogTo(log => logs.Add(log),
        LogLevel.Information);
using var context = new BookDbContext(builder.Options);
//...Здесь идет ваш запрос;
```

Сообщение журнала захватывается и добавляется в logs

Создает DbContext приложения – в данном случае контекст, содержащий данные о книгах

Настраивает уровень журналирования. Уровень Information содержит выполненный SQL-код

По умолчанию исключения не содержат конфиденциальных данных. Этот код включает вывод конфиденциальных данных

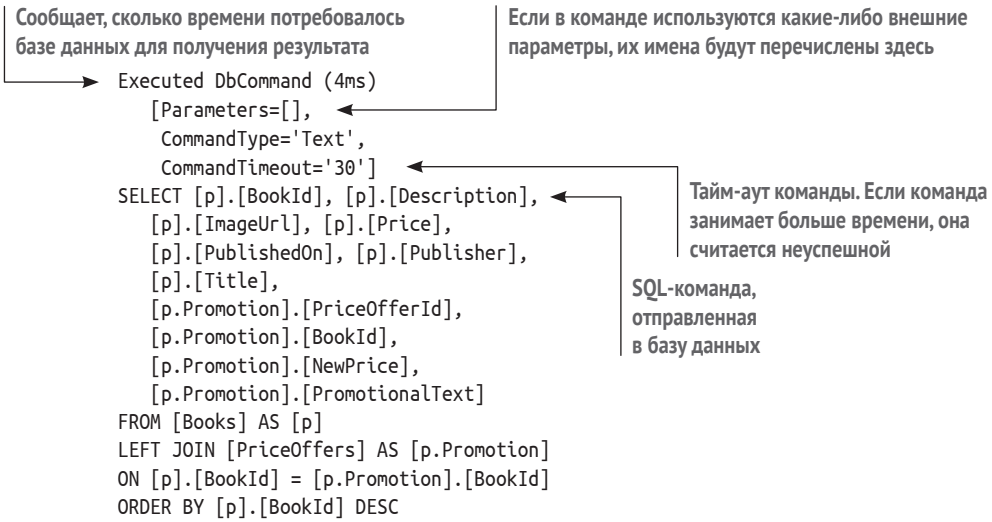
> **ПРЕДУПРЕЖДЕНИЕ** Метод `EnableSensitiveDataLogging` из листинга 14.1 будет включать в журналы любые параметры запросов. Этот метод полезен для отладки, но *не* нужно использовать его в реальном приложении, поскольку параметры могут содержать личные данные, которые не должны записываться в журнал по соображениям безопасности и/или конфиденциальности.

Мы рассмотрели, как получать сообщения журналов EF Core; далее вы увидите, как использовать эту информацию, чтобы обнаружить проблемы с производительностью.

ИЗВЛЕЧЕНИЕ КОМАНД SQL, ОТПРАВЛЯЕМЫХ В БАЗУ ДАННЫХ

EF Core пишет в журналы все свои действия, и эти журналы могут быть полезны. Если задать в своем приложении уровень `Information`, то можно получить полный список команд SQL, генерируемых EF Core и отправляемых в базу данных. В следующем листинге показан пример сообщения `Information`, содержащего SQL-код из контекста варианта приложения Book App из первой или второй части.

Листинг 14.2 Сообщение уровня `Information`, показывающее команду SQL, отправляемую в базу данных




```
Executed DbCommand (4ms)
   [Parameters=[],
    CommandType='Text',
    CommandTimeout='30']
SELECT [p].[BookId], [p].[Description],
   [p].[ImageUrl], [p].[Price],
   [p].[PublishedOn], [p].[Publisher],
   [p].[Title],
   [p.Promotion].[PriceOfferId],
   [p.Promotion].[BookId],
   [p.Promotion].[NewPrice],
   [p.Promotion].[PromotionalText]
FROM [Books] AS [p]
LEFT JOIN [PriceOffers] AS [p.Promotion]
ON [p].[BookId] = [p.Promotion].[BookId]
ORDER BY [p].[BookId] DESC
```

Те из вас, кому нравится работать с SQL, могут скопировать SQL-код из сообщений журнала и запустить его в анализаторе запросов. Microsoft SQL Server Management Studio (SSMS) позволяет запустить запрос и просмотреть план его выполнения, который сообщает, из чего состоит каждая часть запроса, и относительные затраты на каждую часть. Для других баз данных есть свои анализаторы запросов, например Query Analyzer для MySQL и plprofiler для PostgreSQL.

14.3 Часть 3: методы устранения проблем с производительностью

В оставшейся части этой главы приводится список хороших и плохих паттернов EF Core для доступа к базе данных. Они предназначены для того, чтобы научить вас тому, что может повысить или снизить производительность. Их можно использовать в качестве справочника по вопросам производительности базы данных. Этот раздел состоит из четырех частей:

- *хорошие паттерны EF Core* – паттерны из разряда «Применять всегда», которыми вы, возможно, захотите воспользоваться. Они не надежны, но обеспечивают неплохой старт вашему приложению;
- *плохие паттерны запросов к базе данных – антипаттерны* или паттерны, которые не следует применять, потому что, как правило, они создают неэффективные SQL-запросы;
- *плохие программные паттерны* – антипаттерны, значительно замедляющие выполнение кода доступа к базе данных;
- *паттерны масштабируемости* – методы, помогающие базе данных обрабатывать большое количество операций доступа к данным.

В главе 15 приводится пример подходов к настройке производительности, показанных в этой главе. Она начинается с настройки команд EF Core в приложении Book App, а затем переходит к более глубоким методам, таким как замена использования EF Core на чистые SQL-команды и изменение структуры базы данных для повышения производительности. Глава 16 выводит обсуждение на новый уровень, используя CQRS и базу данных Cosmos DB, обладающую отличной производительностью и масштабируемостью.

14.4 Использование хороших паттернов позволяет приложению хорошо работать

Хотя я и не поклонник настройки производительности на раннем этапе, я все же обращаю внимание на аспекты производительности всех паттернов, которые использую. Глупо использовать паттерн, который с самого начала приведет к низкой производительности. Многие из паттернов и практик, описанных в этой книге, действительно влияют на производительность или упрощают ее настройку. Вот список паттернов, помогающих решить проблемы с производительностью, которые я всегда применяю со старта проекта:

- использование метода `Select` для загрузки только нужных столбцов;
- использование разбиения на страницы и/или фильтрации результатов поиска для уменьшения количества загружаемых строк;
- понимание того, что использование отложенной загрузки повлияет на производительность базы данных;
- добавление метода `AsNoTracking` к запросам с доступом только на чтение;
- использование асинхронных версий команд EF Core для улучшения масштабируемости;
- поддержание кода доступа к базе данных изолированным/слабосвязанным, чтобы он был готов к настройке производительности.

14.4.1 Использование метода Select для загрузки только нужных столбцов

В разделе 2.4 вы узнали о четырех способах загрузки связанных данных, в одном из которых использовалась команда LINQ `Select`. Для запросов к базе данных, получающих информацию из нескольких таблиц, метод `Select` часто обеспечивает самый эффективный код (см. раздел 14.5.1 для получения дополнительной информации о снижении обращений к базе данных). Этот процесс показан на рис. 14.5.

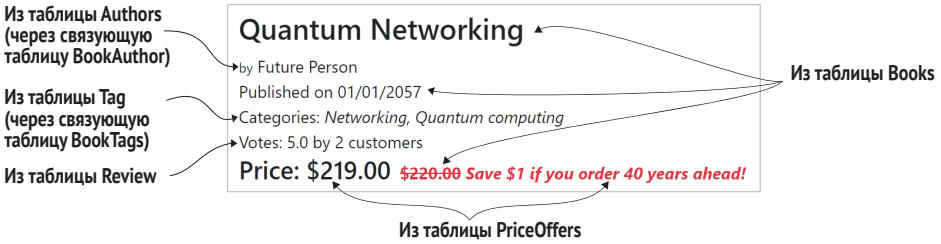


Рис. 14.5. Выборочные запросы обеспечивают наиболее эффективный доступ к базе данных, где конечный результат – данные столбцов из нескольких таблиц

Создание выборочного запроса и использование DTO требует больше усилий, чем использование немедленной загрузки с методом `Include` (см. раздел 2.4.1), но помимо более высокой производительности доступа к базе данных здесь есть и другие преимущества, например уменьшение связанности данных.

> **СОВЕТ** В разделе 6.1.9 описывается, как использовать AutoMapper, чтобы автоматизировать построение запроса `Select` и тем самым ускорить разработку.

14.4.2 Использование разбиения по страницам и/или фильтрации результатов поиска для уменьшения количества загружаемых строк

Поскольку в запросах EF Core используются команды LINQ, иногда можно забыть, что один запрос может вернуть тысячи или миллионы строк. Запрос, который отлично работает на стадии разработки, где в таблице может быть всего несколько строк, может ужасно работать в рабочей системе, у которой гораздо больший набор данных. Вы должны применять команды, которые ограничат объем данных, возвращаемых пользователю. Вот типичные подходы:

- *разбиение на страницы* – вы возвращаете пользователю ограниченный набор данных (скажем, 100 строк) и предоставляете пользовательские команды для перехода по «страницам» данных (см. раздел 2.7.3);

- *фильтрация* – если у вас много данных, то пользователь оценит функцию поиска, которая вернет подмножество данных (см. раздел 2.7.2).

Помните, что нельзя писать открытые запросы, такие как `context.Books.ToList()`, потому что вы можете быть шокированы, если такой запрос будет запущен в рабочей системе, особенно если вы пишете код для сайта по продаже книг наподобие Amazon.

14.4.3 Понимание того, что отложенная загрузка влияет на производительность базы данных

Отложенная загрузка (см. раздел 2.4.4) – это подход, позволяющий загружать связи при обращении к ним. Она присутствует в EF6.x и была добавлена в EF Core 2.1. Проблема состоит в том, что отложенная загрузка пагубно влияет на производительность доступа к базе данных, и после того как вы использовали ее в своем приложении, для ее замены может потребоваться довольно много работы.

Это тот случай, когда вы встаете на путь низкой производительности, о чем можете пожалеть. Когда я понял последствия отложенной загрузки в EF6.x, то больше не использовал ее. Конечно, в некоторых случаях она может облегчить разработку, но каждая отложенная загрузка будет добавлять еще одно обращение к базе данных. Учитывая, что первый антипаттерн производительности, который я перечислил, – это «Не минимизировать количество обращений к базе данных» (раздел 14.5.1), то наличие у вас большого количества отложенных загрузок сделает запрос медленным.

14.4.4 Добавление метода AsNoTracking к запросам с доступом только на чтение

Если вы читаете классы сущностей напрямую и не собираетесь их обновлять, стоит включить в запрос метод `AsNoTracking` (см. раздел 6.1.2). Он указывает EF Core не создавать снимок отслеживания для загруженных сущностей, что позволяет сэкономить время и память. Кроме того, это помогает при сохранении данных, поскольку сокращает объем работы, выполняемой методом `DetectChanges` (см. раздел 14.6.2).

Запрос из листинга 14.3 – пример запроса, для которого метод `AsNoTracking`, выделенный жирным шрифтом, улучшает производительность. Простой тест на производительность загрузки 100 книг с отзывами и авторами в главе 6 показал, что использование этого метода было на 50 % быстрее – крайний случай, потому что в запросе было 5000 отзывов; меньшее количество связей даст меньше выигрыша в производительности. Смотрите табл. 6.1.

Листинг 14.3 Использование метода AsNoTracking для повышения производительности запроса

```
var result = context.Books
    .Include(r => r.Reviews)
    .AsNoTracking()
    .ToList();
```

Возвращает класс сущности Book и коллекцию классов сущностей Review

Добавление метода AsNoTracking указывает EF Core не создавать снимок отслеживания, что экономит время и память

Если вы используете выборочный запрос, где результат отображается в DTO, а DTO не содержит классов сущностей, не нужно добавлять метод `AsNoTracking`. Но если DTO содержит класс сущности, то добавление этого метода поможет.

14.4.5 Использование асинхронной версии команд EF Core для улучшения масштабируемости

Рекомендуемая Microsoft практика для приложений ASP.NET – использовать асинхронные методы везде, где это возможно (см. раздел 5.10). Эта практика улучшает масштабируемость сайта, высвобождая поток. Пока метод ожидает ответа от базы данных, этот поток может выполнять запрос другого пользователя.

В настоящее время использование async/await имеет небольшие затраты на производительность, поэтому для приложений, которые обрабатывают несколько одновременных запросов, таких как сайт, нужно использовать данный подход. В разделе 14.7.2 эта тема освещается более подробно.

14.4.6 Поддержание кода доступа к базе данных изолированным/слабосвязанным

Как я уже сказал ранее, для начала я рекомендую запустить код EF Core без настройки производительности – но вы должны быть готовы сделать этот код быстрее, если потребуется. Чтобы добиться изолированности / слабой связанности, убедитесь, что ваш код:

- *находится в четко обозначенном месте* (*изолирован*). Изолирование каждого обращения к базе данных в отдельный метод позволяет найти код, влияющий на производительность;
- *содержит только код доступа к базе данных* (*слабая связанность*). Мой совет: не смешивайте код доступа к базе данных с другими частями приложения, такими как пользовательский интерфейс или API. Таким образом, вы можете изменять этот код, не беспокоясь о других проблемах, не связанных с базой данных.

На протяжении всей книги вы встречали множество примеров такого подхода. В главе 2 вы познакомились с паттерном *Объект запроса* (см. раздел 2.6), а в главе 4 было продемонстрировано использование отдельного проекта для хранения кода доступа к базе данных для

бизнес-логики (см. раздел 4.4.4). Эти паттерны упрощают настройку производительности кода доступа к базе данных, так как у вас есть четко определенный участок кода, с которым нужно работать.

14.5 Антипаттерны производительности: запросы к базе данных

Предыдущие паттерны стоит использовать постоянно, но вы все равно столкнетесь с проблемами, требующими настройки LINQ. EF Core не всегда создает самые эффективные команды SQL, иногда из-за отсутствия надлежащей трансляции SQL, а иногда из-за того, что написанный вами код LINQ не так эффективен, как вы думали.

В этом разделе представлены некоторые антипаттерны производительности, которые влияют на время, необходимое для передачи данных в базу данных и обратно. Я использую негативный термин «антипаттерны», потому что это то, что вы ищете, – места, где можно улучшить код. Вот список потенциальных проблем с указанием способов их устранения, причем наиболее вероятные проблемы, с которыми вы столкнетесь, перечислены первыми:

- отсутствие минимизации количества обращений к базе данных;
- отсутствие индексов для свойства, по которому вы хотите выполнить поиск;
- использование не самого быстрого способа загрузки отдельной сущности;
- перенос слишком большой части запроса данных на сторону приложения;
- расчеты вне базы данных;
- использование неоптимального SQL-кода в запросе LINQ;
- отсутствие предварительной компиляции часто используемых запросов.

14.5.1 Антипаттерн: отсутствие минимизации количества обращений к базе данных

Если вы запрашиваете сущность из базы данных со связанными с ней данными, у вас есть четыре способа загрузки этих данных: выборочная, немедленная, явная и отложенная загрузки. Хотя все способы дают одинаковый результат, их эффективность сильно отличается. Основное различие сводится к количеству выполняемых ими отдельных обращений к базе данных; чем больше отдельных обращений вы совершаете, тем больше времени займет доступ к вашей базе данных.

Начиная с EF Core 3.0 способ по умолчанию для обработки любых коллекций в запросе заключался в загрузке коллекции с базовой сущностью. `context.Books.Include(b => b.Reviews)`, например, загрузит

сущность Book и связанные сущности Review за один запрос к базе данных. Выборочные запросы и запросы с немедленной загрузкой загружают коллекции из базы данных за один вызов. В примерах запросов в следующих фрагментах кода происходит только по одному обращению к базе данных:

```
var bookInclude = context.Books.Include(b => b.Reviews).First();

var bookSelect = context.Books.Select(b => new
{
    b.Title,
    Reviews = b.Reviews.ToList()
}).First();
```

С другой стороны, явная или отложенная загрузка потребует двух обращений к базе данных. Чтобы увидеть влияние различных подходов на производительность, загрузим сущность Book с Reviews, BookAuthor и Authors (два автора), используя следующие варианты: выборочная загрузка / немедленная загрузка, немедленная загрузка с AsSplitQuery (см. раздел 6.1.4) и явная/отложенная загрузка. Результаты показаны в табл. 14.1.

Таблица 14.1 Сравнение четырех способов загрузки данных, из которого видно, что чем больше выполняется обращений к базе данных, тем больше времени занимает запрос

Тип запроса	Количество обращений к базе данных	Время в EF Core 5 (мс) / %
Выборочная загрузка и немедленная загрузка	1	1.95 / 100 %
Немедленная загрузка с AsSplitQuery	4	2.10 / 108 %
Явная и отложенная загрузки	6	4.40 / 225 %

ПРЕДУПРЕЖДЕНИЕ Запросы, включающие несколько коллекций с большим количеством записей, не будут работать должным образом, если вы используете подход с запросом по умолчанию. Загрузка сущности с тремя коллекциями, каждая из которых содержит 100 записей, вернет 100 * 100 * 100 = 1 000 000 строк. В таких случаях следует добавить к своему запросу метод AsSpiltQuery. Подробную информацию об этом см. в разделе 6.1.4.

ПРИМЕЧАНИЕ Цифры в табл. 14.1 настолько отличались от первого издания книги, что я запустил старый код, чтобы проверить результаты, и EF Core 2.1 был намного медленнее. EF Core 3.0 улучшил загрузку коллекций, а NET 5 сократил время, необходимое для доступа к базе данных SQL Server.

Благодаря улучшениям в EF Core различия между вышеуказанными вариантами загрузки стали меньше, но неоднократное обращение

к базе данных по-прежнему требует затрат. Таким образом, правило состоит в том, чтобы попытаться создать один запрос LINQ, который получит все необходимые данные за одно обращение к базе данных. Выборочные запросы являются наиболее эффективными, если вам нужны только определенные свойства; в противном случае лучше использовать немедленную загрузку с методом `Include`, если вы хотите, чтобы сущность со своими связями отслеживалась для обновления.

14.5.2 Антипаттерн: отсутствие индексов для свойства, по которому вы хотите выполнить поиск

Если вы планируете выполнить поиск по свойству, которое не является ключом (EF Core автоматически добавляет индекс к первичным, внешним или альтернативным ключам), то добавление индекса к этому свойству улучшит производительность поиска и сортировки. Добавить индекс к свойству несложно; см. раздел 6.9.

Обновление индекса требует небольших затрат производительности, если значение свойства (столбец) изменяется, но часто затраты на обновления намного меньше, чем повышение производительности при сортировке или фильтрации по этому свойству. Тем не менее добавление индексов лучше всего работает, если у вас много записей для сортировки/фильтрации по свойству, а скорость чтения важнее времени обновления.

14.5.3 Антипаттерн: использование не самого быстрого способа загрузки отдельной сущности

Когда я изучал EF Core, то думал, что лучший способ загрузить отдельную сущность – это использовать метод EF Core `Find`. Я использовал его, пока не встретил Рика Андерсона, который работает в Microsoft. Он использует метод `FirstOrDefault`. Я спросил почему, и он сказал, что так быстрее. Тогда я измерил производительность, и он оказался прав.

В табл. 14.2 приводится хронометраж для каждого из методов, которые можно было бы использовать для загрузки отдельной сущности через первичный ключ.

Таблица 14.2 Время, затраченное на чтение одной книги разными способами. Хронометраж взят на основе среднего времени, затраченного на загрузку 1000 книг. Обратите внимание, что здесь две версии загрузки с помощью метода `Find`

Метод	Время	Соотношение
`context.Books.Single(x => x.BookId == id)`	175 мкс	100 %
`context.Books.First(x => x.BookId == id)`	190 мкс	109 %
`context.Find<Book>(id)` (сущность не отслеживается)	610 мкс	350 %
`context.Find<Book>(id)` (сущность уже отслеживается)	0.5 мкс	0.3 %

ПРИМЕЧАНИЕ Я не смог найти никакой существенной разницы в производительности между синхронными и асинхронными версиями или методами `First` или `FirstOrDefault`, которые я показываю.

Из таблицы видно, что метод `Single` (и `SingleOrDefault`) был самым быстрым для доступа к базе данных, кроме того, он лучше, чем `First`, поскольку выдает исключение, если оператор `Where` возвращает несколько результатов. Кроме того, методы `Single` и `First` позволяют применять в запросе методы `Include`.

Вы должны использовать метод `Find`, если сущность отслеживается в контексте, в этом случае `Find` будет сверхбыстрым; см. последнюю строку табл. 14.2. Это быстрый метод, потому что сначала он сканирует отслеживаемые сущности, и если находит нужную сущность, то возвращает ее без доступа к базе данных. Обратная сторона этого поиска состоит в том, что этот метод работает медленнее, если сущности нет в контексте.

ПРИМЕЧАНИЕ Метод `Find` вернет отслеживаемую сущность, которая еще не была добавлена или обновлена в базе данных. Я использую эту возможность в обработчике параллелизма (см. листинг 15.11) для пересчета кешированного значения с использованием имени нового автора, которое еще не было записано в базу данных.

14.5.4 Антипаттерн: перенос слишком большой части запроса данных на сторону приложения

Очень просто написать код LINQ, который перемещает вычисление из базы данных в приложение. Часто это сильно влияет на производительность. Начнем с простого примера.

Листинг 14.4 Две команды LINQ с разным временем выполнения

Этот запрос будет работать хорошо, поскольку часть, касающаяся оператора Where, будет выполняться в базе данных

Этот запрос будет работать плохо, поскольку будут возвращены все книги (что требует времени), а после этого часть, касающаяся оператора Where, будет выполняться в приложении

```
context.Books.Where(p => p.Price > 40).ToList();
context.Books.ToList().Where(p => p.Price > 40);
```

Хотя большинство сразу заметит ошибку в листинге 14.4, вполне возможно, что код, подобный тому, что содержится в этом листинге, можно каким-то образом скрыть. Поэтому если вы обнаружите запрос, на выполнение которого уходит много времени, проверьте его составные части.

В EF Core 3 появилось одно существенное изменение: разрешалось вычислять на клиенте (см. раздел 2.3) только последний вызов `Select` в запросе. Эта ситуация вызвала проблемы после обновления до EF Core 3, но она лишь выявила запросы LINQ, которые выполнялись медленно. После этого изменения, если EF Core не сможет преобразовать ваш запрос в команды базы данных, вы получите исключение `could not be translated (не может быть преобразовано)`, таким образом перехватывается много неверных LINQ-запросов. Далее это исключение гласит:

```
… or switch to client evaluation explicitly by inserting a call
to 'AsEnumerable', 'AsAsyncEnumerable', 'ToList', or 'ToListAsync'
(… либо явно используйте вычисление на клиенте, добавив вызов 'AsEnumerable',
'AsAsyncEnumerable', 'ToList' или 'ToListAsync')
```

Полезное сообщение, но иногда EF Core выдает его, потому что вы не совсем правильно написали запрос LINQ. Агрегатам LINQ (т. е. `Sum`, `Max`, `Min` и `Average`; см. раздел 14.5.5) для работы требуется версия типа, допускающая значение NULL, и если вы не предоставите ее, то получите исключение `could not be translated`. См. «Агрегатам требуется значение null (кроме count)» в разделе 6.1.7. Поэтому, прежде чем добавлять `'AsEnumerable'`, `'AsAsyncEnumerable'` и т. д., следует проверить, как преобразовать запрос в команды базы данных.

14.5.5 Антипаттерн: вычисления вне базы данных

Одна из причин, по которой приложение Book App работает быстро, заключается в том, что мы перенесли часть вычислений в базу данных, в частности количество отзывов и среднюю оценку из `Reviews`. Если бы мы не перенесли эти вычисления в базу данных, то приложение работало бы, но было бы медленным, особенно при сортировке или фильтрации по средней оценке.

Как правило, в базу данных не получится перенести много вычислений, но те, что получится, могут иметь большое значение, особенно если вы хотите отсортировать или отфильтровать вычисленное значение. Вот несколько примеров того, что можно сделать:

- подсчитать элементы в навигационном свойстве коллекции, например `Book.Reviews`. Это полезный подход, если вам нужно количество, но не нужно содержимое коллекции;
- суммировать значения в коллекции, например суммировать стоимости всех `LineItem` в `Order`. Это полезный подход, если вы хотите отсортировать заказы по сумме.

ПРИМЕЧАНИЕ Смотрите раздел 6.1.7, где говорится о командах LINQ, требующих особого внимания для преобразования LINQ-запросов в команды базы данных.

14.5.6 Антипаттерн: использование неоптимального SQL-кода в LINQ-запросе

Иногда вы знаете что-то о своих данных, что позволяет вам создать фрагмент SQL-кода, который будет лучше, чем сгенерированный EF Core. Но в то же время вы не хотите лишиться простоты создания запросов, используя EF Core и LINQ. Есть несколько способов добавить вычисления SQL к обычным запросам LINQ:

- *добавьте пользовательские функции в запросы LINQ*. Скалярная пользовательская функция (см. раздел 10.1) возвращает одно значение, которое можно присвоить свойству в запросе, тогда как табличная функция возвращает данные, как если бы они были получены из таблицы. В разделе 15.3 я использую скалярную функцию для построения списка имен авторов книги;
- *создайте представление SQL в своей базе данных, где есть команды SQL для вычисления значений*. Отобразите класс сущности в это представление (см. раздел 7.9.3), а затем примените запросы LINQ к этому отображенному классу. Такой подход дает возможность добавить некий сложный SQL-код внутри представления при использовании LINQ для доступа к этим данным;
- *используйте SQL-методы EF Core,* `FromSqlRaw` *и* `FromSqlInterpolated`. Эти методы позволяют применять SQL для обработки первой части запроса. Далее вы можете использовать другие команды LINQ, например сортировки и фильтрации, но прочтите раздел 11.5, чтобы узнать об ограничениях методов `FromSqlRaw` и `FromSqlInterpolated`;
- *настройте свойство как вычисляемый столбец*. Используйте этот подход, если вычисление данного свойства можно выполнить с помощью других свойств/столбцов в классе сущности и/или команд SQL. (См. примеры в листинге 10.7, а дополнительные сведения о вычисляемых столбцах см. в разделе 10.2.)

Ясно, что нужно понимать и писать SQL-код, но если вы можете это делать, то эти методы проще, чем использовать библиотеки, работающие с SQL напрямую, например ADO.NET или Dapper (см. раздел 11.5.5).

14.5.7 Антипаттерн: отсутствие предварительной компиляции часто используемых запросов

Когда вы впервые выполняете запрос EF Core, он компилируется и кешируется, поэтому при повторном использовании скомпилированный запрос можно найти в кеше, что позволяет избежать его повторной компиляции. Но есть (небольшие) затраты на этот поиск в кеше, которые может обойти метод EF Core `EF.CompiledQuery`. Если у вас есть запрос, который вы часто используете, стоит попробовать этот метод, но не думаю, что предварительно скомпилированные запросы силь-

но улучшат производительность. Другая проблема заключается в том, что у предварительно скомпилированных запросов есть некоторые ограничения, что может затруднить их использование:

- можно использовать скомпилированный запрос только в том случае, если команда LINQ не создается динамически, когда части запроса добавляются или удаляются. Например, метод `BookListFilter` динамически создает команду LINQ с помощью оператора switch, поэтому вы не сможете превратить ее в скомпилированный запрос;
- запрос возвращает один класс сущности – `IEnumerable<T>` или `IAsyncEnumerable<T>`, – поэтому вы не можете связывать объекты запроса в цепочку, как мы это делали в главе 2.

Метод `EF.CompiledQuery` позволяет хранить скомпилированный запрос в статической переменной, которая устраняет необходимость поиска в кеше. У LINQ-запросов могут быть переменные в методах, и значения этих переменных передаются с помощью DbContext, как показано в следующем листинге.

Листинг 14.5 Создание скомпилированного запроса и сохранение его в статической переменной

Вы определяете статическую функцию для хранения скомпилированного запроса – в данном случае функцию с двумя параметрами и типом возвращаемого запроса

```
private static Func<EfCoreContext, int, Book>
    _compiledQueryComplex =          f
    EF.CompileQuery(
        (EfCoreContext context, int i) =>
        context.Books
            .Skip(i)
            .First()
        );
```

Определяет запрос, который следует хранить как скомпилированный

Ожидает DbContext, один или два параметра для использования в запросе и возвращаемый результат (класса сущности или IEnumerable <TEntity>)

Метод `EF.CompiledQuery` предназначен для получения и компиляции конкретного запроса. В случае запроса на книгу нужно будет создать отдельный скомпилированный запрос для каждого параметра фильтрации и сортировки, чтобы можно было скомпилировать каждый из них, как показано ниже:

- запрос на книги, без фильтрации и сортировки;
- запрос на книги, фильтрация по средней оценке, без сортировки;
- запрос на книги, фильтрация и сортировка по средней оценке;
- запрос на книги, фильтрация по средней оценке, сортировка по дате публикации.

Это полезный метод, но лучше всего применять его, когда запрос, производительность которого вы хотите настроить, является стабильным, поскольку, возможно, вам придется поработать, чтобы привести запрос к правильной форме, дабы он соответствовал методу `EF.CompiledQuery`.

14.6 Антипаттерны производительности: операции записи

Теперь, когда вы познакомились с антипаттернами производительности, которые применяются к запросам, рассмотрим антипаттерны, применяемые к операциям записи. Эти проблемы представляют собой смесь паттернов, которые приводят к снижению производительности либо базы данных, либо времени вычислений в приложении. Сначала я перечислил наиболее вероятные проблемы:

- неоднократный вызов метода `SaveChanges`;
- слишком большая нагрузка на метод `DetectChanges`;
- `HashSet<T>` не используется для навигационных свойств коллекции;
- использование метода `Update`, когда нужно изменить только часть сущности;
- проблема с запуском: использование одного большого контекста DbContext.

14.6.1 Антипаттерн: неоднократный вызов метода SaveChanges

Если в базу данных нужно добавить много информации, есть два варианта:

- *добавить одну сущность и вызвать метод* `SaveChanges`. Если вы сохраняете 10 сущностей, вы вызываете метод `Add` с последующим вызовом метода `SaveChanges` 10 раз;
- *добавить все экземпляры сущностей и в конце вызвать метод* `SaveChanges`. Чтобы сохранить 10 сущностей, вызовите метод `Add` 10 раз (или, лучше, один раз метод `AddRange`), после чего в конце один раз вызовите метод `SaveChanges`.

Вариант 2 – вызов метода `SaveChanges` только один раз – *намного* быстрее, как видно из табл. 14.3, потому что EF Core сгруппирует несколько операций записи данных для выполнения на серверах баз данных, допускающих такой подход, например SQL Server. В результате подобный подход сгенерирует код SQL, который более эффективен при записи нескольких элементов в базу данных. В табл. 14.3 показана разница во времени для двух способов записи 100 новых сущностей в базу данных SQL Server.

Разница между двумя способами сохранения нескольких сущностей может быть большой. В примере в табл. 14.3, где метод `SaveChanges` вызывается 100 раз (слева), затраченное время более чем в 15 раз больше, чем при однократном вызове этого же метода (справа).

Некоторая потеря в производительности при использовании подхода «один за раз» происходит из-за дополнительного обращения к базе данных. Подход «все сразу» задействует возможности пакетной

обработки EF Core, что дает SQL-код, который хорошо работает при добавлении большого количества данных в базу. Для получения более подробной информации по этой теме см. http://mng.bz/ksHg.

> **ПРИМЕЧАНИЕ** Кроме того, не рекомендуется вызывать метод `SaveChanges` после каждого изменения: что произойдет, если на полпути что-то пойдет не так? Рекомендуется выполнить все свои добавления, обновления и удаления, а затем вызвать этот метод в конце. Таким образом, вы знаете, что все ваши изменения были применены к базе данных или, если произошла ошибка, ни одно из изменений применено не было.

Таблица 14.3 Сравнение вызова метода `SaveChanges` после добавления каждой сущности и добавления всех сущностей и последующего вызова метода `SaveChanges` в конце. Вызов этого метода в конце примерно в 15 раз быстрее, чем вызов этого же метода после каждого метода `Add`

Один за раз	Все сразу (пакетом в SQL Server)

`for (int i = 0; i < 100; i++)`
`{`
`  context.Add(new MyEntity());`
`  context.SaveChanges();`
`}`
	`for (int i = 0; i < 100; i++)`
	`{`
	`  context.Add(new MyEntity());`
	`}`
	`context.SaveChanges();`

Общее время = 160 мс	Общее время = 9 мс

14.6.2 Антипаттерн: слишком большая нагрузка на метод DetectChanges

Каждый раз, когда вы вызываете метод `SaveChanges`, по умолчанию он выполняет метод `ChangeTracker.DetectChanges` внутри DbContext приложения, чтобы определить, были ли обновлены какие-либо отслеживаемые сущности. (Подробности см. в разделе 11.3.3.) Время, необходимое для выполнения метода `DetectChanges`, зависит от того, сколько отслеживаемых сущностей загружено, т. е. от количества сущностей, которые вы читаете без методов `AsNoTracking` или `AsNoTrackingWithIdentityResolution` (см. раздел 6.1.2), которые не реализуют интерфейс `INotifyPropertyChanged` (см. раздел 11.4.2).

В табл. 14.4 показано время, затраченное на разные уровни отслеживаемых сущностей. В данном случае сущности небольшие и у них несколько свойств; если бы отслеживаемые сущности были более сложными, то время было бы больше.

У такого рода проблем есть разные решения в зависимости от дизайна приложения. Вот способы решить такую проблему с производительностью:

- Нужно ли загружать все эти сущности как отслеживаемые? Если метод `SaveChanges` занимает много времени, возможно, вы забыли использовать методы `AsNoTracking` и `AsNoTrackingWithIdentityResolution` при выполнении запросов с доступом только на чтение?

Таблица 14.4 Время, затраченное методом SaveChanges, который содержит вызов метода DetectChanges.Detect, чтобы сохранить одну сущность для разных уровней отслеживаемых сущностей. Обратите внимание, что отслеживаемые сущности, используемые в этой таблице, небольшие

Количество отслеживаемых сущностей	Сколько времени потребовалось методу SaveChanges	Насколько медленнее?
0	0,2 мс	Н/Д
100	0,6 мс	В 2 раза медленнее
1000	2,2 мс	В 11 раз медленнее
10000	20 мс	В 100 раз медленнее

- Можно ли разбить большую вставку на мелкие части? Я делаю это в главе 15, где создаю класс для создания больших наборов тестовых данных для тестов производительности. В этом классе я веду запись партиями по ~ 700 книг и использую новый экземпляр DbContext приложения, чтобы не было отслеживаемых сущностей.
- Если нужно загрузить много сущностей для изменения, рассмотрите возможность изменения классов сущностей для использования стратегии отслеживания изменений `INotifyPropertyChanged`. Это изменение требует изменения кода классов сущностей, чтобы добавить `INotifyPropertyChanged` и настроить стратегию отслеживания изменений для классов сущностей (см. раздел 11.4.2). В результате сущности будут сообщать обо всех изменениях в EF Core, и методу `DetectChanges` не нужно сканировать загруженные вами сущности на предмет изменений.

14.6.3 Антипаттерн: HashSet<T> не используется для навигационных свойств коллекции

В разделе 6.2.2 вы узнали, что при вызове метода `Add` для добавления новой сущности в базу данных EF Core выполняет ряд шагов, чтобы убедиться, что все связи настроены правильно. Один из этапов, выполняемых EF Core, называемый ссылочной фиксацией, проверяет, отслеживаются ли уже какие-либо сущности в добавленной сущности. (Подробности см. в разделе 6.2.2.)

С точки зрения производительности этап ссылочной фиксации может стоить вам времени, потому что он должен сравнивать все отслеживаемые сущности, которые используются в добавленной сущности и ее связях. Трудно получить надежный хронометраж, потому что первые несколько применений в DbContext приложения происходят медленно, но вот некоторые наблюдения:

- когда вы загружаете навигационные свойства коллекции, например, с помощью метода `Include`, `HashSet<T>` для коллекций работает быстрее, чем навигационные свойства коллекции типов `ICollection<T>` / `IList<T>`. Допустим, добавление сущности

с 1000 сущностей в навигационное свойство коллекции заняло на 30 % больше времени с `ICollection<T>`, чем при использовании `HashSet<T>`, потому что в `HashSet<T>` легче обнаруживать экземпляры;

- чем больше отслеживаемых сущностей одного типа, обнаруженных в сущности (и ее связях), было добавлено, тем больше времени уйдет на их проверку. Падение производительности трудно измерить, но оно кажется небольшим. Однако если у вас проблемы с методом `Add`, который отнимает много времени, стоит выполнить проверку на предмет наличия большого количества отслеживаемых сущностей. Отчасти это и может быть причиной медлительности при вызове метода `Add`;
- как было сказано в разделе 2.1.3, недостаток использования `HashSet<T>` состоит в том, что он не гарантирует порядок записей в коллекции. Поэтому при использовании возможностей EF Core 5 для сортировки записей в методе `Include` нельзя использовать `HashSet<T>`.

14.6.4 Антипаттерн: использование метода Update, когда нужно изменить только часть сущности

EF Core отлично подходит для обнаружения изменений отдельных свойств в классе сущности, используя метод `DetectChanges.Detect`. Если вы измените одно свойство, например дату публикации книги, а затем вызовете метод `SaveChanges`, то метод `DetectChanges.Detect` обнаружит это изменение, и EF Core создаст SQL-код для обновления этого единственного столбца в правильной строке таблицы Books.

С другой стороны, если вы используете метод `Update` сущности `Book`, то все свойства будут помечены как изменившиеся, SQL-код увеличится, и потребуется (немного) больше времени, чтобы его выполнить. Метод `Update` следует использовать только тогда, когда изменилась вся сущность; см. раздел 11.3.4, где приводится пример.

14.6.5 Антипаттерн: проблема при запуске – использование одного большого DbContext

В первый раз, когда создается DbContext приложения, на это уходит некоторое время, возможно, несколько секунд. Причин такой медлительности много, но одна из них – EF Core необходимо просканировать все классы сущностей в DbContext приложения, чтобы настроиться и создать модель базы данных, к которой вы хотите получить доступ. Обычно это небольшая проблема, потому что после запуска приложения информация о конфигурации и модели базы данных кешируется EF Core. Но если ваше приложение постоянно запускается и останавливается – скажем, в бессерверной архитектуре (https://martinfowler.com/articles/serverless.html), – то время запуска может иметь значение.

Можно ускорить первоначальное создание DbContext приложения, уменьшив количество классов сущностей, которые он включает в себя. Единственный разумный способ сделать это – создать несколько DbContext, каждый из которых охватывает подмножество таблиц в базе данных. В разделе 13.4.8 описывается разделение базы данных на несколько DbContext на основе подхода DDD с ограниченными контекстами. На рис. 14.6 показано, как разделить большую базу данных между DbContext нескольких приложений.

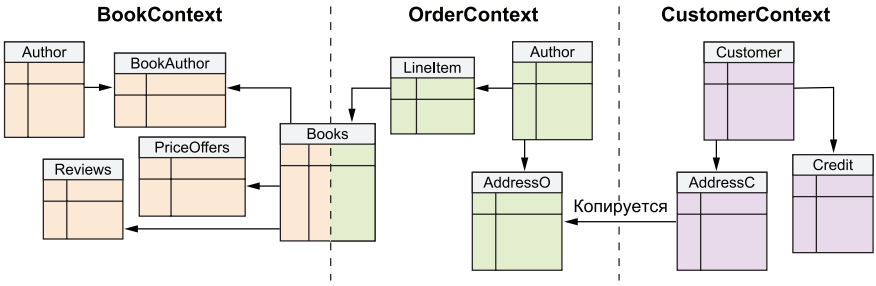


Рис. 14.6 Большую базу данных можно разделить на DbContext нескольких приложений. В данном случае база данных разделена по бизнес-направлениям. Если нужно минимизировать затраты на запуск приложения, то можно создать определенные DbContext для каждого приложения, содержащие только те сущности, к которым приложение должно получить доступ

На рис. 14.6 база данных разбивается на DbContext разных приложений на основе предметных областей, что может быть подходящим вариантом для некоторых приложений. Если вы создаете небольшие автономные приложения, например в бессерверной архитектуре или микросервисах (https://martinfowler.com/articles/microservices.html), то можно создать DbContext приложения, включающий только сущности и таблицы, специфичные для каждого приложения.

14.7 Паттерны производительности: масштабируемость доступа к базе данных

Масштабируемость приложения (количество одновременных обращений, которые может обработать приложение) – большая тема. Даже при ограничении масштабируемости доступа к базе данных есть над чем подумать. Проблемы с масштабируемостью обычно нельзя отследить до плохо написанного фрагмента кода, потому что масштабируемость больше зависит от дизайна. В этом разделе:

- использование пулов для снижения затрат на создание нового DbContext приложения;
- добавление масштабируемости с незначительным влиянием на общую скорость;

- улучшение масштабируемости базы данных за счет упрощения запросов;
- масштабирование сервера базы данных;
- выбор правильной архитектуры для приложений, которым требуется высокая масштабируемость.

14.7.1 Использование пулов для снижения затрат на создание нового DbContext приложения

Если вы создаете приложение ASP.NET Core, то EF Core предоставляет метод `AddDbContextPool<T>`, заменяющий обычный метод `AddDbContext<T>`. Метод `AddDbContextPool<T>` использует внутренний пул экземпляров DbContext приложения, которые он может переиспользовать. Этот метод уменьшает время ответа приложения, когда у вас много коротких запросов.

Но имейте в виду, что в некоторых ситуациях использовать его не следует. Когда вы передаете данные на основе HTTP-запроса, например идентификатор выполнившего вход пользователя, не следует использовать пул DbContext, поскольку он будет использовать неправильный идентификатор пользователя в некоторых экземплярах DbContext приложения. Пулы DbContext просты в использовании, и в этом листинге показана обновленная регистрация контекста `EfCoreContext` в приложении Book App.

Листинг 14.6 Использование `AddDbContextPool` для регистрации DbContext приложения

Вы используете базу данных SQL Server, но пулы работают с любым провайдером базы данных

Регистрируем DbContext приложения с помощью метода AddDbContextPool<T>

```
services.AddDbContextPool<EfCoreContext>(
    options => options.UseSqlServer(connection,
    b => b.MigrationsAssembly("DataLayer")));
```

Поскольку мы используем миграции в многослойной архитектуре, необходимо сообщить провайдеру базы данных, в какой сборке находится код миграции

Влияют ли пулы DbContext на масштабируемость приложения, зависит от типа имеющегося у вас параллельного трафика. Но вы должны добиться по крайней мере небольшого улучшения скорости, так как метод `AddDbContextPool<T>` будет быстрее возвращать экземпляры DbContext.

14.7.2 Добавление масштабируемости с незначительным влиянием на общую скорость

В разделе 14.4.5 было сказано, что нужно использовать асинхронные версии методов доступа к базе данных в приложении, которые должны обрабатывать несколько одновременных запросов, потому что

async/await высвобождает поток, позволяющий обрабатывать другие запросы, в то время как асинхронная часть ожидает ответа от базы данных (см. рис. 5.8). Но использование асинхронного метода вместо обычного синхронного метода добавляет небольшие накладные расходы к каждому вызову. В табл. 14.5 приведены показатели производительности для нескольких типов доступа к базе данных.

Различия между синхронной и асинхронной версиями в табл. 14.5 невелики, однако есть разница: для медленных запросов требуется асинхронный режим, поскольку он высвобождает поток на длительное время. Но тот факт, что самые быстрые запросы имеют наименьшее различие между синхронной и асинхронной версиями, говорит о том, что использование асинхронной версии не оказывает заметного влияния на небольшие запросы. В целом при использовании async/await вы получаете много преимуществ и мало недостатков.

Таблица 14.5 Производительность для разных типов доступа к базе данных, возвращающих книги, с использованием синхронной и асинхронной версий. База данных содержит 1000 книг

Тип доступа к базе данных	Количество обращений к базе данных	Синхронная версия	Асинхронная версия	Разница
Только чтение книги, простая загрузка	1	0,7 мс	0,8 мс	112 %
Чтение книги, связи с немедленной загрузкой	1	9,7 мс	13,7 мс	140 %
Чтение книги, связи с немедленной загрузкой + сортировка и фильтрация	1	10,5 мс.	14,5 мс.	140 %

14.7.3 Повышение масштабируемости базы данных за счет упрощения запросов

Создание команд SQL, которые не требуют больших затрат на сервере базы данных (они просты в выполнении и возвращают минимальный объем данных), минимизирует нагрузку на базу данных. Настройка производительности ключевых запросов, чтобы они были простыми и возвращали только необходимые данные, не только увеличивает скорость приложения, но и помогает с масштабируемостью базы данных.

14.7.4 Вертикальное масштабирование сервера базы данных

С переходом на использование облачных баз данных можно повысить производительность своей базы данных одним нажатием кнопки (и кредитной картой!). Вариантов так много (в Azure насчитывается более 50 вариантов для SQL Server), что нетрудно сбалансировать производительность и затраты.

14.7.5 Выбор правильной архитектуры для приложений, которым требуется высокая масштабируемость

В разделе 5.2 подробно описано, что у веб-приложения может быть несколько экземпляров для обеспечения большей масштабируемости. Запуск нескольких экземпляров приложения полезен для производительности программного обеспечения и вычислений, но если все экземпляры приложения обращаются только к одной базе данных, это не обязательно способствует масштабируемости базы данных.

Хотя производительности программного обеспечения и вычислений обычно являются проблемным местом в масштабируемости, в случае с приложениями, которые предъявляют высокие требования к базе данных, дополнительные экземпляры веб-приложения не сильно помогут. На данном этапе нужно подумать о других архитектурах. Один из подходов, называемый *шардингом*, распределяет данные по нескольким базам данных, что может сработать для определенных типов многопользовательских приложений. В главах 15 и 16 мы изучим два архитектурных подхода – кеширование и CQRS, – которые улучшают производительность и масштабируемость.

Поскольку большинство приложений читают базу данных больше, чем пишут в нее, архитектура CQRS может помочь в повышении производительности базы данных. Кроме того, выделив запросы с доступом только на чтение к нереляционной базе данных под названием Cosmos DB, можно упростить репликацию баз данных с доступом только на чтение, что повысит пропускную способность базы данных. Я реализую такую архитектуру, используя CQRS из главы 16, с впечатляющим приростом производительности.

Резюме

- Не настраивайте производительность слишком рано; пусть ваше приложение сначала поработает должным образом. Но попробуйте спроектировать его таким образом, чтобы, если вам понадобится улучшить производительность позже, было легче найти и исправить код доступа к базе данных.
- Настройка производительности не бесплатна, поэтому нужно решить, какие проблемы с производительностью стоят усилий разработчиков, чтобы устранить их.
- Сообщения журналов EF Core могут помочь вам определить код доступа к базе данных, в котором есть проблемы с производительностью.

- Убедитесь, что все стандартные паттерны или методы, которые вы используете при написании своего приложения, хорошо работают. В противном случае вы столкнетесь с проблемами производительности с самого начала.
- Избегайте любых антипаттернов производительности базы данных (обращений к базе данных, которые плохо работают) или исправляйте их.
- Если масштабируемость является проблемой, попробуйте простые улучшения, но высокая масштабируемость может потребовать фундаментального переосмысления архитектуры приложения.
- В главе 15 приведен пример применения рекомендаций из этой главы для повышения производительности приложения Book App.

Для читателей, знакомых с EF6:

- некоторые проблемы с производительностью в EF6.x, например использование метода `AddRange` при повторных вызовах метода `Add`, были исправлены в EF Core.

15 Мастер-класс по настройке производительности запросов к базе данных

В этой главе рассматриваются следующие темы:

- четыре подхода к настройке производительности запросов в EF Core;
- сравнение прироста производительности, обеспечиваемого каждым подходом;
- извлечение хороших практик из каждого подхода, чтобы использовать их в своих приложениях;
- оценка навыков и затрат на разработку, необходимых для реализации каждого подхода;
- что такое масштабируемость базы данных и как ее улучшить.

В главе 14 содержится много информации о том, как настроить производительность приложения. В этой главе и частично в главе 16 вы увидите, насколько быстро можно настроить отображение книг в варианте приложения Book App из третьей части. Эта информация познакомит вас с различными способами настройки производительности приложения EF Core; каждый подход предполагает баланс между лучшей производительностью и дополнительным временем разра-

ботки. Изучив ряд подходов, вы будете готовы решить, что вам нужно в собственных приложениях.

Мы будем применять различные подходы к настройке производительности, которые будут постепенно увеличивать быстродействие приложения Book App, требуя все больше и больше усилий по разработке для достижения прироста производительности. Хотя конкретный код для прироста производительности из этого приложения, возможно, и не подойдет вашему приложению, каждое изменение производительности использует разную методологию, чтобы вы могли выбрать вариант, который вам подходит.

В этой главе основное внимание уделяется запросам с доступом только на чтение, которые часто являются основными проблемными местами в приложениях. Что касается записи в базу данных, см. раздел 14.6.

15.1 Настройка тестового окружения и краткое изложение четырех подходов к повышению производительности

Прежде чем мы сможем оптимизировать производительность приложения, нам понадобится несколько наборов данных для тестирования. Иногда данные поступают из существующего приложения, в котором есть проблемы с производительностью, или, возможно, ваша проектная команда либо менеджеры задали цели по производительности. Но для повышения производительности нужны тестовые данные, представляющие реальные данные, с которыми вы можете столкнуться в реальном мире.

Что касается третьей части, я обратился к Manning Publications (издателю этой книги), который предоставил набор реальных данных, содержащий около 700 книг. На рис. 15.1 показана информация об этой книге в приложении Book App. (Щелкнув по названию книги, вы попадаете на страницу с подробными данными и изображением обложки.)

> **ПРИМЕЧАНИЕ** Вы можете самостоятельно запустить этот пример, скачав репозиторий GitHub для этой книги (http://mng.bz/XdlG), а затем выбрав ветку `Part3`. Проект BookApp.UI содержит приложение ASP.NET Core. Запустив его, вы увидите раздел домашней страницы «Things to Do» для ссылки на информацию о настройке приложения для отображения четырех подходов, использованных в этой главе.

В данной главе используются четыре подхода для настройки производительности:

- *хороший LINQ* – использует тот же подход, что и в разделе 2.6, и следует предложениям из главы 14. Этот подход – наша базовая производительность;
- *LINQ + пользовательские функции* – объединяет LINQ с пользовательскими функциями SQL (см. раздел 10.1) для перемещения конкатенации имен авторов и тегов в базу данных;
- *SQL + Dapper* – создает необходимые команды SQL, а затем использует Dapper для выполнения этого SQL-кода для чтения данных;
- *LINQ + кеширование* – предварительный расчет некоторых дорогостоящих частей запроса, например `NumStars` у `Review` (называемых *оценками*).

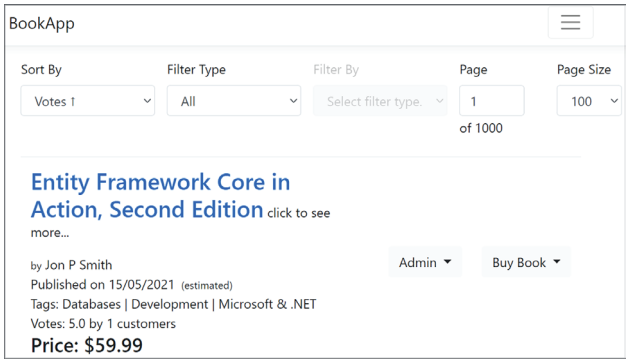


Рис. 15.1 Обновленный BookApp.UI с использованием реальных данных о книгах от Manning Publications. На этом рисунке показан пример приложения Book App с использованием реальных данных о книгах, предоставленных издательством, которые были продублированы, чтобы в базе было 100 000 книг, а также более полумиллиона отзывов

Чтобы предоставить более требовательный набор данных для тестирования этих подходов, используйте `BookGenerator` приложения Book App, чтобы скопировать первые 700 книг. Для тестов в этой главе я использовал 100 000 книг. В табл. 15.1 показан полный список данных в базе данных.

Таблица 15.1 Тестовые данные, используемые в этой главе для тестирования четырех подходов к производительности

Таблица	Books	Review	BookAuthor	Authors	BookTags	Tags
Количество строк	100 000	546 023	156 958	868	174 405	35

В этой главе мы сравним производительность этих четырех подходов для трех разных запросов для отображения книг. Эти три запроса варьируются от простого запроса с сортировкой по дате до сложного запроса с сортировкой по оценкам. На рис. 15.2 показано время, затраченное на каждый подход к каждому из трех запросов, с использованием тестовых данных, подробно описанных в табл. 15.1.

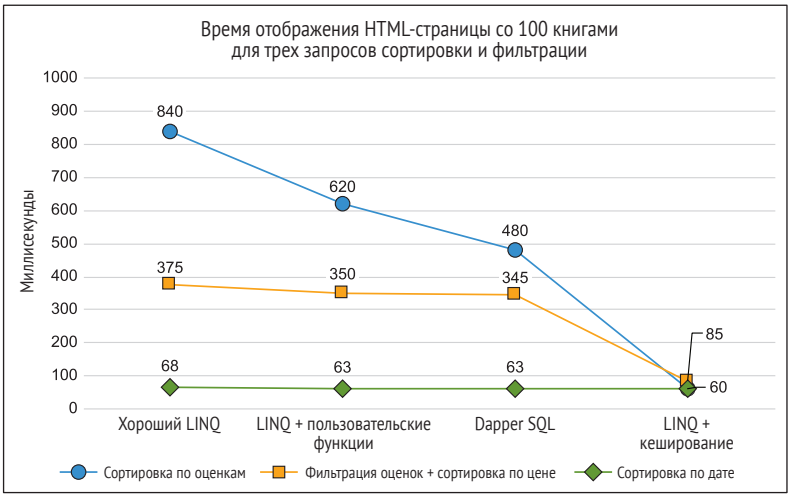


Рис. 15.2 На диаграмме показано время, необходимое для отображения страницы, содержащей 100 книг для трех разных типов сортировки и фильтрации. База данных содержит 100 000 книг и полмиллиона отзывов (см. табл. 15.1 для получения полной информации). Я использовал свой ПК и SQL Server localdb, работающий на нем; все запросы асинхронные

Вот подробное описание трех типов запросов, показанных на рис. 15.2:

- *сортировка по оценкам* – сортировка по средней оценке, которая рассчитывается как среднее значение свойства `NumStars` в отзывах, связанных с книгой. Этот запрос показывает, что сортировка по средней оценке, которую пользователи часто будут использовать, сильно различается по быстродействию во всех четырех подходах по причинам, описанным в разделах, посвященных каждому подходу;
- *фильтрация по оценке + сортировка по цене* – отфильтровывает все книги, набравшие меньше 4 баллов по средней оценке (около 3000 книг), а затем выполняет сортировку по цене. Этот запрос показывает, что первые три подхода занимают одинаковое количество времени. Кешированная версия быстрая, потому что средняя оценка предварительно вычисляется и у нее есть индекс SQL;
- *сортировка по дате* – сортировка по дате публикации книги. Это сортировка по известному свойству, у которого есть индекс SQL. Все подходы обеспечивают хорошую производительность с небольшими различиями между подходом *хороший LINQ* и подходами *SQL + пользовательские функции* и *Dapper + SQL*.

Хотя эти четыре подхода применяются к приложению Book App, они определяют четыре общих подхода к настройке производительности запросов к базе данных в EF Core. В объяснении каждого подхода подробно описывается применение улучшений производитель-

ности приложения, после чего приводятся выводы, чтобы вы могли решить, будет ли это работать в вашем приложении с EF Core.

15.2 Хороший LINQ: использование выборочного запроса

Этот подход близок к запросу, который мы создали в главе 2. Самое замечательное в этом подходе заключается в том, что он прост: запрос использует только LINQ для построения этой версии, тогда как версии LINQ + SQL и Dapper требуют, чтобы вы использовали SQL-код, а кешированный SQL требует серьезных усилий, чтобы заставить его работать.

Во-первых, должен сказать, что текущий LINQ-запрос достаточно быстрый. Здесь всего 700 книг; на сортировку голосов и отображение 100 книг уходит около 70 мс. Причина состоит в том, что в запросе из главы 2 уже используются некоторые хорошие практики. Я не упоминал о них в главе 2, потому что эта глава была в начале книги, но теперь мы можем подробно изучить данный запрос.

> **ПРИМЕЧАНИЕ** Если вы загрузите и запустите приложение Book App из ветки `Part3`, то увидите SQL-код, генерируемый каждым из подходов. Выберите подход и тип фильтрации и сортировки, а затем щелкните на пункт меню *Logs*. Вы увидите SQL-код, использованный в выполненном вами запросе.

В следующем листинге показана часть запроса, которая собирает все необходимые данные и комментарии к различным частям, что делает его хорошим LINQ-запросом.

Листинг 15.1 Метод `MapBookToDto`, который выбирает, что показывать в запросе на отображение книги

```
public static IQueryable<BookListDto>
    MapBookToDto(this IQueryable<Book> books)
{
    return books.Select(p      => new BookListDto
    {
        BookId                 = p.BookId,
        Title                  = p.Title,
        PublishedOn            = p.PublishedOn,
        EstimatedDate          = p.EstimatedDate,
        OrgPrice               = p.OrgPrice,
        ActualPrice            = p.ActualPrice,
        PromotionText          = p.PromotionalText,
```

Хорошая практика: загружайте только те свойства, которые вам нужны

Хорошая практика: используйте индексированные свойства для сортировки/фильтрации (в данном случае – ActualPrice)

Хорошая практика: не загружайте все связи, а только те части, которые вам нужны

```
        AuthorsOrdered          = string.Join(", ",
            p.AuthorsLink
                .OrderBy(q      => q.Order)
                .Select(q       => q.Author.Name)),
        TagStrings              = p.Tags
            .Select(x           => x.TagId).ToArray(),
        ReviewsCount            = p.Reviews.Count(),
        ReviewsAverageVotes     =
            p.Reviews.Select(y =>
                (double?)y.NumStars).Average(),
        ManningBookUrl          = p.ManningBookUrl
    });
}
```

Хорошая практика: ReviewsCount и ReviewsAverageVotes рассчитываются в базе данных

Теперь посмотрим, что происходит в методе расширения `MapBookToDto`, чтобы можно было понять и применить эти хорошие практики в собственных приложениях.

ЗАГРУЖАЕМ ТОЛЬКО СВОЙСТВА, НЕОБХОДИМЫЕ ДЛЯ ЗАПРОСА

Можно было бы загрузить всю сущность `Book`, но это означало бы загрузку ненужных данных. Данные по книгам от Manning Publications содержат большие строки с содержанием книги, технологиями, о которых в ней рассказывается, и т. д. Однако для отображения книги эти данные не нужны, и их загрузка замедлит выполнение запроса, поэтому мы не загружаем их.

В соответствии с рекомендацией, согласно которой не нужно настраивать производительность на ранних этапах, можно начать с простого запроса, который считывает классы сущностей, а производительность настроить позже. В приложении Book App было очевидно, что запрос на отображение книги является ключевым, особенно в отношении сортировки по голосам, поэтому я начал с выборочного запроса. Но в вашем случае, если запрос выполняется медленно и вы загружаете весь класс сущности, подумайте об использовании метода `Select` для загрузки только необходимых свойств.

НЕ ЗАГРУЖАЙТЕ ВСЕ СВЯЗИ – ТОЛЬКО ТЕ ЧАСТИ, КОТОРЫЕ ВАМ НУЖНЫ

Есть много способов загрузить связи, включая немедленную, явную и отложенную загрузки. Проблема состоит в том, что эти три подхода к чтению связей загружают весь класс сущности. Как правило, его не нужно загружать целиком.

В листинге 15.1 видно, что коллекция `AuthorLink` используется для выбора только имени автора, что минимизирует количество данных, возвращаемых из базы. Точно так же используется Tags, чтобы вернуть лишь массив `TagId`. Итак, чтобы повысить производительность запроса, если вам нужны данные из связей, попробуйте извлечь определенные части из них. Есть идея и получше – по возможности перенесите вычисления в базу данных, о чем я расскажу далее.

ПО ВОЗМОЖНОСТИ ПЕРЕНЕСИТЕ ВЫЧИСЛЕНИЯ В БАЗУ ДАННЫХ

Если вам нужна хорошая производительность, особенно для сортировки или фильтрации значений, которые нуждаются в вычислении, гораздо лучше, чтобы вычисления производились внутри базы данных. Здесь есть два преимущества:

- данные, используемые в вычислениях, никогда не покидают базу данных, поэтому в приложение нужно отправлять меньше данных;
- вычисленное значение можно использовать для сортировки или фильтрации, поэтому можно выполнить запрос к базе данных в одной команде.

Если, например, вы не вычислили значение `ReviewsAverageVotes` в базе данных, то нужно будет прочитать *все* свойства `Reviews` `NumStars` и `BookId` и вычислить значение `ReviewsAverageVotes` для каждой книги. Только тогда вы сможете решить, какие сущности `Book` выбрать для отображения. Этот процесс будет медленным и потребует много памяти, потому что нужно будет прочитать все отзывы из базы данных, а затем вычислить среднюю оценку на стороне приложения, прежде чем прочитать сущности `Book`, которые нужно отобразить.

Должен сказать, что правильное выполнение таких расчетов было неочевидным! Когда я писал первое издание этой книги, мне не удалось получить правильный запрос значения `ReviewsAverageVotes`, и потребовалось задать вопрос на странице отзывов EF Core на GitHub, чтобы получить правильный ответ. В разделе 6.1.8 я рассматриваю команды LINQ, которые должны быть написаны определенным образом, чтобы они работали.

ПО ВОЗМОЖНОСТИ ИСПОЛЬЗУЙТЕ ПРОИНДЕКСИРОВАННЫЕ СВОЙСТВА ДЛЯ СОРТИРОВКИ И ФИЛЬТРАЦИИ

В первой части я применил специальное предложение к `Book`, добавив класс сущности `PriceOffer`. Я сделал это не только потому, что хотел показать, как работают связи «один к одному», но и потому, что использование класса сущности `PriceOffer` сделало очевидными мои действия. Обратная сторона этого подхода состоит в том, что запрос должен включать код для поиска класса сущности `PriceOffer`. Следующий фрагмент кода взят из версии метода `MapBookToDto` для первой части:

```
ActualPrice = book.Promotion == null
    ? book.Price
    : book.Promotion.NewPrice,
PromotionPromotionalText =
    book.Promotion == null
    ? null
    : book.Promotion.PromotionalText,
```

Этот код оказывает два отрицательных эффекта на сортировку по цене: код LINQ преобразуется в SQL-оператор `JOIN` для поиска необязательной строки PriceOffers, что требует времени, и вы не можете добавить индекс SQL к этому вычислению. В третьей части приложение Book App перешло на использование предметно-ориентированного проектирования (DDD), поэтому можно было добавлять или удалять ценовые предложения с помощью методов доступа в сущности `Book` (см. раздел 13.4.2). Методы доступа скрывают бизнес-логику промоакции, а это означает, что свойство `ActualPrice` всегда содержит цену, по которой продается книга. Изменение кода, чтобы не использовать сущность `PriceOffer`, удаляет оператор `JOIN`, и вы можете добавить индекс для столбца ActualPrice в базе данных, что значительно улучшает сортировку по цене.

Поэтому если нужно запросить какие-то данные, особенно если вы сортируете или фильтруете их, попробуйте предварительно вычислить данные в своем коде. Или используйте постоянный вычисляемый столбец (см. раздел 10.2), если свойство вычисляется на основе других свойств и столбцов в том же классе сущности, например `[TotalPrice] AS (NumBook*BookPrice)`. Таким образом, вы получите значительное улучшение при сортировке или фильтрации благодаря индексу SQL в этом столбце.

15.3 LINQ + пользовательские функции: добавляем SQL в код LINQ

При подходе *Хороший LINQ* код LINQ, формирующий операции чтения для отображения книг, возвращает коллекции имен авторов и `TagId`, потому что авторов и тегов может быть много. До EF Core 3.0 эти коллекции читались с помощью дополнительного запроса для каждой коллекции, поэтому чтение 100 книг только с именем автора создавало 101 обращение к базе данных (одно для основного запроса, а затем по одному для каждой книги для имени автора) и занимало около 230 мс.

Начиная с EF Core 3.0 этот запрос был сокращен до одного обращения к базе данных за счет возврата нескольких строк для каждой книги и дополнительных столбцов, чтобы убедиться, что строки находятся в правильном порядке. При большом количестве сущностей `Book`, `Author` и `TagId` завершающая часть SQL-кода, производимого подходом *Хороший LINQ*, который используется для показа книг, с упорядочением по умолчанию (по убыванию `BookId`) выглядит так:

```
SELECT [t].[BookId],…
-- other parts of the SQL
ORDER BY [t].[BookId] DESC
   ,[t0].[Order]
   ,[t0].[BookId], [t0].[AuthorId], [t0].[AuthorId0]
   ,[t2].[BookId], [t2].[TagId0], [t2].[TagId]
```

Я не буду объяснять различные таблицы и столбцы в `ORDER BY` (можно увидеть весь SQL-запрос, запустив приложение Book App и щелкнув пункт меню «Logs»), но видно, что здесь много параметров `ORDER BY`. Оказывается, если добавить LINQ-запрос с сортировкой по средней оценке в верхней части `ORDER BY`, то производительность начнет падать. Это одна из причин, почему отображение книг с использованием данного подхода настолько медленное.

ПРИМЕЧАНИЕ Прежде чем вы скажете, что наличие всех этих параметров `ORDER BY` – плохой SQL-код, могу ответить, что без этого кода запрос занял бы примерно вдвое больше времени и перешел бы от одного обращения к базе данных к пяти отдельным обращениям. Изменение в EF Core 3.0 улучшило большую часть, но не все (см. раздел 6.1.4) запросы, содержащие коллекции.

Некоторое время назад я нашел на Stack Overflow SQL-код, объединявший серию строк в одну строку внутри базы данных. В разделе 14.5.6 я описал четыре способа улучшить LINQ-запрос, предоставив SQL-код, настроенный под конкретную ситуацию. В данном случае я использовал скалярную пользовательскую функцию для доступа к этому коду, как показано в следующем фрагменте:

```
CREATE FUNCTION AuthorsStringUdf (@bookId int)
RETURNS NVARCHAR(4000)
AS
BEGIN
-- Thanks to https://stackoverflow.com/a/194887/1434764
DECLARE @Names AS NVARCHAR(4000)
SELECT @Names = COALESCE(@Names + ', ', '') + a.Name
FROM Authors AS a, Books AS b, BookAuthor AS ba
WHERE ba.BookId = @bookId
      AND ba.AuthorId = a.AuthorId
      AND ba.BookId = b.BookId
ORDER BY ba.[Order]
RETURN @Names
END
```

ПРИМЕЧАНИЕ Следует добавлять SQL-код только в том случае, если у вас есть код, который делает что-то лучше, чем EF Core. Простое добавление SQL-кода, аналогичного тому, что было бы создано в EF Core, не улучшит производительность.

Чтобы использовать код `AuthorsStringUdf` и `TagsStringUdf` для объединения `TagId`, мне пришлось определить его (см. раздел 10.1) и добавить пользовательские функции в базу данных, отредактировав миграцию (см. раздел 9.5.2). Затем мне нужно было создать новое отображение из сущности `Book` в DTO для отображения книг, как показано в следующем листинге. Смотрите строки с комментариями, где вызываются две пользовательские функции.

Листинг 15.2 Метод MapBookUdfsToDto, использующий пользовательские функции для конкатенации имен Name и Tag

```
public static IQueryable<UdfsBookListDto>
    MapBookUdfsToDto(this IQueryable<Book> books)
{
    return books.Select(p => new UdfsBookListDto
    {
        BookId = p.BookId,
        Title = p.Title,
        PublishedOn = p.PublishedOn,
        EstimatedDate = p.EstimatedDate,
        OrgPrice = p.OrgPrice,
        ActualPrice = p.ActualPrice,
        PromotionText = p.PromotionalText,
        AuthorsOrdered = UdfDefinitions
            .AuthorsStringUdf(p.BookId),
        TagsString = UdfDefinitions
            .TagsStringUdf(p.BookId),
        ReviewsCount = p.Reviews.Count(),
        ReviewsAverageVotes =
            p.Reviews.Select(y =>
                (double?)y.NumStars).Average(),
        ManningBookUrl = p.ManningBookUrl
    });
}
```

Обновленный метод MapBookToDto, теперь он называется MapBookUdfsToDto

Для AuthorsOrdered и TagsString задаются значения в виде строк из пользовательских функций

При изменении метода расширения `MapBookToDto` для использования пользовательских функций `AuthorsStringUdf` и `TagsStringUdf` каждая книга возвращает только одну строку, и здесь нет оператора `ORDER BY`, кроме упорядочивания по умолчанию – по убыванию `BookId`. Это изменение не сильно влияет на несортированный показ 100 книг (улучшая его на несколько миллисекунд), но большое влияние оказывает на сортировку по средней оценке, которая снижается с 840 мс при подходе *Хороший LINQ* до 620 мс при подходе *LINQ + SQL* – улучшение составляет около 25 %.

15.4 SQL + Dapper: написание собственного SQL-кода

Окончательный подход – отказаться от EF Core и написать собственный SQL-запрос. Если вы хотите это сделать, то вам понадобится библиотека, которая может выполнить SQL-код. Лучшее, что я нашел, – это Dapper (о нем рассказывается в разделе 11.5.4). Дело в том, что чистый SQL может быть лучше, чем EF Core.

Я изучил SQL-код, созданный EF Core, немного покопался и нашел одно место, где я мог бы улучшить его. Оказывается, можно выполнить сортировку по параметру в команде `SELECT`: https://stackoverflow.

com/a/38750143/1434764. Согласно сообщению, приведенному на этой странице, «оператор ORDER BY вычисляется после оператора SELECT (это означает, что можно использовать вычисляемый столбец из SELECT), в отличие от операторов WHERE или FROM, которые вычисляются перед SELECT и поэтому не могут ссылаться на псевдонимы столбцов в SQL Server».

EF Core не использует эту функцию, поэтому его SQL-код вычисляет среднюю оценку дважды: один раз в SELECT и еще раз в ORDER BY. Мои тесты показали, что вычисление средней оценки только один раз значительно улучшило производительность в запросе с сортировкой по оценкам, поэтому я приступил к переписыванию различных функций сортировки, фильтрации и разбиения по страницам, используемых в приложении Book App, которые включали выбор и объединение строк SQL для формирования правильного SQL-запроса. Преобразование функций LINQ в SQL было довольно сложным. На рис. 15.3 показана блок-схема, демонстрирующая, как устроен SQL-запрос.

ПРИМЕЧАНИЕ Созданный мной SQL-код использует две пользовательские функции, применяемые в подходе *LINQ + пользовательские функции*; в противном случае все было бы медленнее, чем в этом подходе. Если вы хотите увидеть код, который создает и выполняет SQL, то его можно найти на странице http://mng.bz/n2Q2.

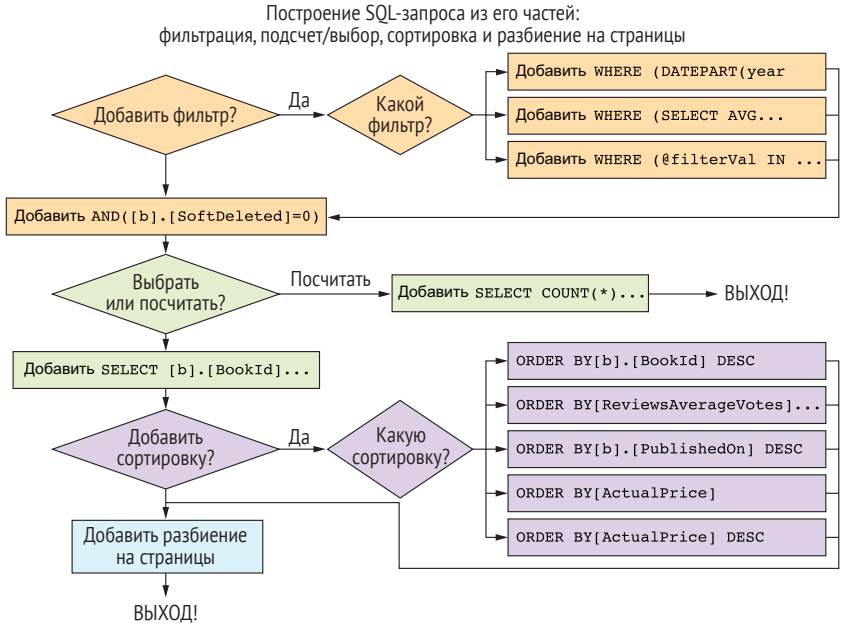


Рис. 15.3 Код Dapper состоит из серии конкатенаций строк, которые формируют окончательный SQL-запрос. Этот код не так элегантен, как версия EF Core с четырьмя объектами запроса, но при настройке производительности часто приходится мириться с потерей чистоты исходного кода, чтобы добиться нужной производительности

Улучшение производительности запроса с сортировкой по оценкам впечатляет: версия с Dapper почти в два раза быстрее, чем версия *Хороший LINQ* (Dapper: 480 мс, Хороший LINQ: 840 мс). Но по всем остальным запросам, не включавшим сортировку по голосам, версия с Dapper была ненамного быстрее, чем версия с LINQ, особенно при подходе *LINQ + пользовательские функции*. Чтобы понять этот результат, я рассмотрел простейший запрос – сортировку по дате публикации, – чтобы увидеть, на что тратится время. Рисунок 15.4 разбивает время на три части:

- (внизу) *время базы данных* (важно) – время, необходимое для выполнения SQL-запроса;
- (в центре) *HTML-время* – время, необходимое для отправки HTML-страницы в браузер;
- (вверху) *время приложения* – остальное время, в основном ASP.NET Core.

ПРИМЕЧАНИЕ Хронометраж SQL получен из журналов EF Core. Сюда входит время, которое потребовалось для выполнения на сервере баз данных. Для Dapper я использовал класс Stopwatch, запустив его перед вызовом Dapper и остановив, когда данные были возвращены.

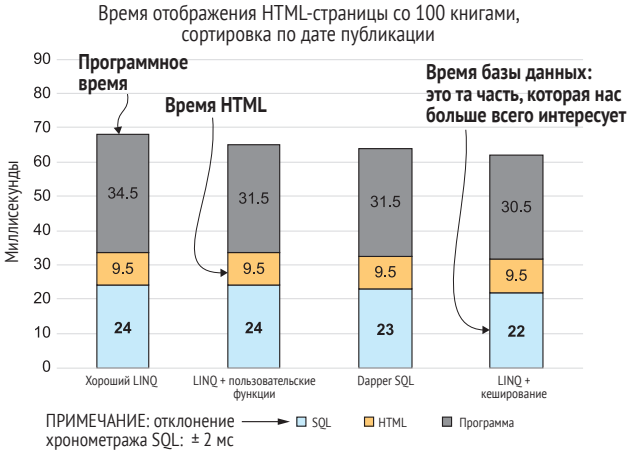


Рис. 15.4 Распределение сортировки по дате публикации книги на странице со 100 книгами. Важная часть, на которую следует обратить внимание, – это нижний хронометраж, показывающий время, необходимое для выполнения SQL-запроса: ± 2 мс для SQL-части, с некоторым разбросом, который я опустил. Остальные части имеют более крупные вариации. Общая вариация составляет 10 мс для версии *Хороший LINQ* и меньше (скажем, 5 мс)

Как видно по рис. 15.4, различия в SQL невелики, и из-за отклонений в ±2 мс, по сути, они одинаковы. Высокая производительность библиотеки Dapper становится все менее важным фактором, когда

выполнение SQL-кода, используемого в запросе, занимает много миллисекунд. А поскольку единственные запросы, для которых нужно настроить производительность, обычно занимают много миллисекунд, 1/2 или 1 мс, сэкономленные Dapper не имеют большого значения. (Помогает и то, что EF Core становится быстрее.)

Вывод из рис. 15.4: преобразовывать медленные запросы в Dapper стоит только в том случае, если можно найти какой-то SQL-код, который лучше, чем тот, который производит EF Core. На создание и отладку сложного отображения книг у меня ушло довольно много времени, и если бы у меня не было другого способа повысить производительность, усилия были бы оправданы. Подход с кешированным SQL (раздел 15.5) обеспечивает гораздо большее улучшение производительности, но и требует гораздо больше усилий.

ПРИМЕЧАНИЕ Для ясности: кроме проблемы с сортировкой по голосам, я не нашел ничего другого в EF Core, что можно было бы улучшить с помощью Dapper, а в EF Core уже была проблема #16038 для решения этого вопроса.

15.5 LINQ + кеширование: предварительное вычисление частей запроса, которое занимает много времени

Последний подход в этой главе – это предварительное вычисление частей запроса, которое занимает много времени, и их сохранение в дополнительных свойствах и столбцах в классе сущности `Book`. Этот метод известен как *кеширование*, или *денормализация*. Кеширование лучше всего работает с данными, которые сложно генерировать, такими как средняя оценка книги. Как было показано на рис. 15.2, кеширование оказывает наибольшее влияние на запрос с сортировкой по оценкам, делая его примерно в 14 раз быстрее, нежели подход *Хороший LINQ*, и в 8 раз быстрее, чем подход с Dapper.

Но, принимая решение об использовании кеширования, также необходимо подумать о том, как часто обновляется кешированное значение и сколько времени требуется для обновления кеша. Если кешируемые данные нередко обновляются, то затраты на обновление кеша могут сместить проблему производительности с выполнения запроса чтения на обновление сущностей. Как вы увидите в разделе 15.5.2, алгоритмы кеширования, используемые в приложении Book App, выполняются быстро, когда дело доходит до обработки обновлений.

Но основная проблема с кешированием заключается в том, что очень сложно обеспечить актуальность кешированных значений. Например, при использовании подхода с кешированием SQL необходимо обновлять кешированное свойство `ReviewsAverageVotes` каж-

дый раз при добавлении, обновлении или удалении отзыва. А что произойдет, если к сущности `Book` применяются два отзыва одновременно или когда обновление кешированного свойства `ReviewsAverageVotes` в базе данных дает сбой? Вот цитата из 1990-х гг., в которой говорится, что обновление кеша всегда было проблемой:

> *В информатике есть только две сложные вещи: инвалидация кеша и придумывание названий переменных.*
>
> – Фил Карлтон (в свою бытность сотрудником Netscape)

Мой опыт показывает, что создать систему кеширования непросто. Я построил систему кеширования для первого издания книги, и она неплохо работала, но теперь я знаю об одной редкой ситуации, в которой не удастся правильно обновить кеш. (Я исправил эту проблему в новой версии для данной книги.)

Изучение SQL-запроса показывает, что кеширование средней оценки (среднее число `NumStars` во всех отзывах, связанных с конкретной сущностью `Book`) повысит производительность для сортировки и фильтрации по средней оценке. На этом можно было бы и остановиться, но кеширование средней оценки и объединения имен авторов обеспечили небольшой прирост производительности для отображения книг (прирост производительности примерно в 5 мс для показа 100 книг).

Добавить систему кеширования – задача нетривиальная. Вот что нужно сделать:

1. Добавить способ отслеживания изменений, влияющих на кешированные значения.
2. Добавить код для обновления кешированных значений.
3. Добавить кешированные свойства в сущность `Book` и предоставить корректный код для обработки одновременных обновлений кешированных значений.
4. Создать запрос на отображение книги, чтобы использовать кешированные значения.

В конце описания этой системы кеширования в разделе 15.5.4 описывается система проверки и восстановления, которая проверяет, правильно ли настроены кешированные значения.

15.5.1 Добавляем способ обнаружения изменений, влияющих на кешированные значения

События предметной области дали хорошие результаты (см. главу 12) при реализации кеширования, поэтому этот подход используется в данном проекте. Одна из положительных особенностей заключается в том, что изменение, инициирующее обновление кешированного значения, сохраняется в той же транзакции, которая обновляет кешированное значение (см. рис. 12.3). В результате к базе данных

применяются оба изменения или, если одно из них будет неудачным, ни одно из обновлений применено не будет. Данный подход предотвращает проблемы с реальными и кешированными данными (что известно как грязный кеш).

Что касается обнаружения изменения свойств или связей, можно воспользоваться тем фактом, что версия приложения Book App для третьей части книги использует DDD. Итак, чтобы обновить два кешированных значения, относящихся к `Review`, можно добавить код в методы доступа сущности `Book`, `AddReview` и `RemoveReview`.

Для кешированного свойства `AuthorsOrdered` мы будем использовать не DDD-подход, инициируя событие предметной области, в котором изменяется имя автора. Этот пример показывает, как обрабатывать события предметной области и кеширование, когда вы не используете DDD.

Чтобы ускорить разработку, мы будем использовать мою библиотеку `EfCore.GenericEventRunner`. Она хорошо протестирована и содержит другие возможности, которые ускорят разработку. Итак, посмотрим, как будет выглядеть код, начиная с событийно-расширенного класса `BookDbContext`, как показано в следующем листинге.

Листинг 15.3 Класс `BookDbContext` обновлен для использования `GenericEventRunner`

```
public class BookDbContext
    : DbContextWithEvents<BookDbContext>
{

    public BookDbContext(
        DbContextOptions<BookDbContext> options,
        IEventsRunner eventRunner = null)
        : base(options, eventRunner)
    { }

    //… Остальная часть BookDbContext обычная, поэтому не указана;
}
```

BookDbContext обрабатывает данные, относящиеся к Book

Вместо наследования от DbContext EF Core используется наследование от класса из библиотеки GenericEventRunner

Внедрение зависимостей предоставит EventRunner из GenericEventRunner. Если значение равно null, то события не используются (полезно для модульных тестов)

Конструктору класса DbContextWithEvents требуется EventRunner

Следующий этап – добавление событий в методы `AddReview` и `RemoveReview`. В листинге ниже показано, как эти методы создают событие.

Листинг 15.4 Сущность `Book` с методами `AddReview` и `RemoveReview`

```
public class Book : EntityEventsBase,
    ISoftDelete
{

        //… Остальной код опущен для ясности;
```

Наследование от EntityEventsBase предоставит методы для отправки события

```
    public void AddReview(int numStars,
        string comment, string voterName)
    {
        if (_reviews == null)
            throw new InvalidOperationException(
                "The Reviews collection must be loaded");

        _reviews.Add(new Review(
            numStars, comment, voterName));

        AddEvent(new BookReviewAddedEvent(numStars,
            UpdateReviewCachedValues));
    }

    public void RemoveReview(int reviewId)
    {
        if (_reviews == null)
            throw new InvalidOperationException(
                "The Reviews collection must be loaded");

        var localReview = _reviews.SingleOrDefault(
            x => x.ReviewId == reviewId);
        if (localReview == null)
            throw new InvalidOperationException(
                "The review was not found.");

        _reviews.Remove(localReview);

        AddEvent(new BookReviewRemovedEvent(localReview,
            UpdateReviewCachedValues));
    }

    private void UpdateReviewCachedValues
        (int reviewsCount, double reviewsAverageVotes)
    {
        ReviewsCount = reviewsCount;
        ReviewsAverageVotes = reviewsAverageVotes;
    }
}
```

Метод AddReview – единственный способ добавить отзыв на эту книгу

Добавляет событие предметной области BookReviewAddedEvent с NumStars нового отзыва

Метод RemoveReview – единственный способ удалить отзыв из этой сущности Book

Предоставляет обработчику событий безопасный способ обновления кешированных значений Review

Добавляет событие предметной области BookReviewAddedEvent с удаленным отзывом

Этот закрытый метод может использоваться обработчиками событий для обновления кешированных значений

Чтобы перехватить изменение имени автора, мы воспользуемся подходом, отличным от DDD, и перехватим установку значения свойства. В этом подходе используется резервное поле, чтобы можно было обнаружить изменение имени автора. Модифицированный класс сущности `Author` показан в следующем листинге.

Листинг 15.5 Сущность `Author`, отправляющая событие при изменении свойства `Name`

```
public class Author : EntityEventsBase
{
    private string _name;
```

Добавление EntityEventsBase предоставит методы для отправки события

Резервное поле для свойства Name, которое EF Core будет читать и записывать

```
    public string Name
    {
        get => _name;
        set
        {
            if (value != _name &&
                AuthorId != default)
                AddEvent(
                    new AuthorNameUpdatedEvent());
            _name = value;
        }
    }

    //… Остальной код опущен для ясности;
}
```

Делаем открытым и переопределяем метод установки свойства, чтобы добавить тестирование и отправку события

Если Name изменилось и это не новый автор, отправляется событие предметной области

Обратите внимание: проверка того, следует ли отправлять событие, включает проверку того, установлен ли первичный ключ `Author`, `AuthorId`. Поскольку класс сущности `Author` не следует DDD-стилю, нельзя быть уверенным в том, как разработчик может создать новый экземпляр этой сущности, поэтому мы добавляем дополнительный тест первичного ключа, чтобы гарантировать, что события отправляются только после обновления свойства `Name`.

15.5.2 Добавление кода для обновления кешированных значений

Теперь мы создадим несколько обработчиков событий для обновления кешированных значений при наступлении соответствующего события предметной области. Эти обработчики событий будут вызываться перед методами `SaveChanges` и `SaveChangesAsync`, поэтому изменения, инициирующие события, и последующие изменения, примененные обработчиками событий, будут сохраняться в одной и той же транзакции. Я покажу два стиля обновления кешированных значений в обработчиках событий:

- быстрые дельта-обновления, которые работают с числовыми изменениями кешированных значений. Например, при получении события `AddReview` обработчик события увеличит значение свойства кеша `ReviewsCount`. Этот вариант работает быстро, но требует тщательного написания кода, чтобы гарантировать получение правильного результата в любой ситуации;
- обычные обновления с повторным вычислением, когда вы выполняете запрос для повторного вычисления кешированного значения. Этот вариант используется для обновления свойства кеша `AuthorsOrdered`.

ОБНОВЛЕНИЕ ЗАПИСАННЫХ ЗНАЧЕНИЙ REVIEW С ИСПОЛЬЗОВАНИЕМ ДЕЛЬТА-ОБНОВЛЕНИЙ

Добавление, обновление или удаление отзывов вызывает определенные события, которые, в свою очередь, вызывают запуск дескриптора события, связанного с каждым типом события. В этом примере мы создадим код обработчика событий, который обновит два кешированных значения, `ReviewsCount` и `ReviewsAverageVotes`, в сущности `Book`. На рис. 15.5 показаны этапы процесса добавления нового отзыва в эту сущность, где уже есть один отзыв.

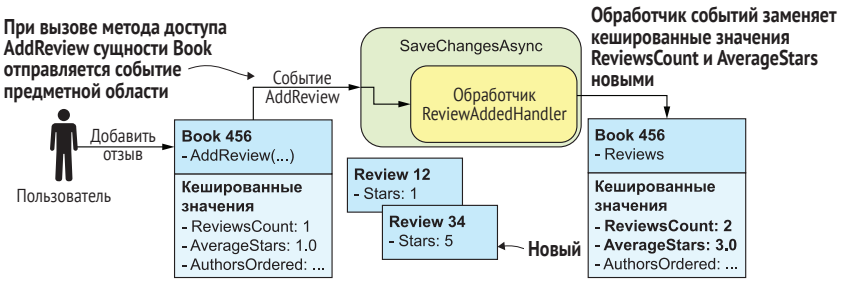


Рис. 15.5 Когда пользователь добавляет новый отзыв, метод доступа `AddReview` создает событие предметной области, которое улавливается `GenericEventRunner`, когда вызываются методы `SaveChanges` или `SaveChangesAsync`. `GenericEventRunner` запускает обработчик `ReviewAddedHandler`, который обновляет кешированные значения отзывов с использованием дельта-обновлений

Основная часть этого процесса находится в обработчике событий, который использует дельта-обновления двух кешированных значений `Review`. В листинге 15.6 показан класс `ReviewAddedHandler`, который библиотека `GenericEventRunner` будет запускать перед вызовом методов `SaveChanges` или `SaveChangesAsync`.

Листинг 15.6 Связывание класса `ReviewAddedHandler` с событием `BookReviewAddedEvent`

Сообщает обработчику событий, что это событие должно вызываться при обнаружении BookReviewAddedEvent

Диспетчер событий предоставляет экземпляр вызывающей сущности и событие

```
public class ReviewAddedHandler :
    IBeforeSaveEventHandler<BookReviewAddedEvent>
{
    public IStatusGeneric Handle(object callingEntity,
        BookReviewAddedEvent domainEvent)
    {
        var book = (Domain.Books.Book) callingEntity;
```

Приводит объект к его фактическому типу Book, чтобы упростить доступ

```
var totalStars = Math.Round(
    book.ReviewsAverageVotes
        * book.ReviewsCount) +
    domainEvent.NumStars;
var numReviews = book.ReviewsCount + 1;

domainEvent.UpdateReviewCachedValues(
    numReviews,
    totalStars / numReviews);

return null;
    }
}
```

Добавляет оценку из нового отзыва

Первая часть этого вычисления определяет общее количество звезд перед добавлением новой оценки

Простое добавление 1 дает новое количество отзывов

Класс сущности предоставляет метод для обновления кешированных значений

Первый параметр – это количество отзывов

Второй параметр предоставляет новое среднее значение NumStars

Возвращение null – быстрый способ сообщить, что обработчик событий всегда успешен

Этот обработчик событий не обращается к базе данных, и он работает быстро, поэтому накладные расходы на обновление кешированных значений `ReviewsCount` и `ReviewsAverageVotes` невелики.

ПРИМЕЧАНИЕ Обработчик событий `RemoveReview` здесь не показан, но работает так же, как и обработчик события `AddReview`.

ОБНОВЛЕНИЕ КЕШИРОВАННОГО ЗНАЧЕНИЯ NAME ПУТЕМ ПОВТОРНОГО ВЫЧИСЛЕНИЯ

Есть много вариантов того, почему автор или его имя в сущности `Book` может измениться. Кто-то мог не упомянуть автора для конкретной книги. Кто-то мог неправильно написать имя автора при добавлении автора (например, *John P Smith* вместо *Jon P Smith*). Для любого из этих изменений затронутая сущность или сущности `Book` должны обновить значение `AuthorsOrdered`. Эта строка не используется при фильтрации или сортировке, но экономит время при отображении имен авторов. В этом примере мы реализуем обновление свойства `Name`, которое требует циклического прохождения всех сущностей `Book`, содержащих сущность `Author`, как показано на рис. 15.6.

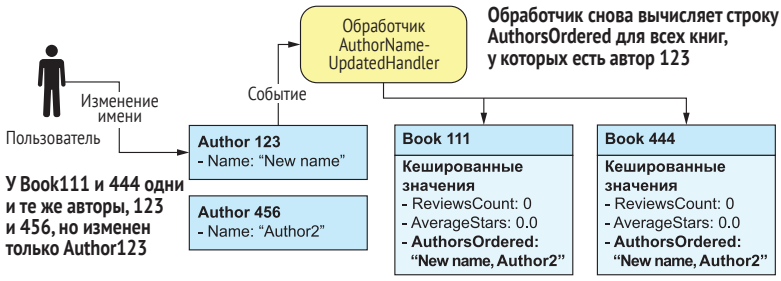


Рис. 15.6 Пользователь с правами администратора изменяет свойство `Name` сущности `Author`, которое используется в двух сущностях `Book`. В этом примере сущности `Book` 111 и 444 связаны с двумя сущностями `Author` – 123 и 456. Для изменения свойства `Name` сущности `Author` 123 требуется, чтобы обработчик событий перебрал все сущности `Book`, в которых используется `Author` 123, и повторно сгенерировал правильную строку `AuthorsOrdered`

В следующем листинге показан класс `AuthorNameUpdatedHandler`. Его вызывает `GenericEventRunner` для обработки события изменения свойства `Name` сущности `Author`. Этот обработчик событий перебирает все сущности `Book`, в которых есть эта сущность `Author`, и повторно генерирует значение `AuthorsOrdered` для каждой сущности `Book`.

Листинг 15.7 Обработчик событий, управляющий изменением свойства `Name`

```
public class AuthorNameUpdatedHandler :
    IBeforeSaveEventHandler<AuthorNameUpdatedEvent>
{
    private readonly BookDbContext _context;

    public AuthorNameUpdatedHandler
        (BookDbContext context)
    {
        _context = context;
    }

    public IStatusGeneric Handle(object callingEntity,
        AuthorNameUpdatedEvent domainEvent)
    {
        var changedAuthor = (Author) callingEntity;

        foreach (var book in _context.Set<BookAuthor>()
            .Where(x => x.AuthorId == changedAuthor.AuthorId)
            .Select(x => x.Book))
        {
            var allAuthorsInOrder = _context.Books
                .Single(x => x.BookId == book.BookId)
                .AuthorsLink.OrderBy(y => y.Order)
                .Select(y => y.Author).ToList();

            var newAuthorsOrdered =
                string.Join(", ",
                    allAuthorsInOrder.Select(x =>
                x.AuthorId == changedAuthor.AuthorId
                    ? changedAuthor.Name
                    : x.Name));

            book.ResetAuthorsOrdered(newAuthorsOrdered);
        }

        return null;
    }
}
```

Сообщает диспетчеру событий, что это событие должно вызываться при обнаружении AuthorNameUpdatedEvent

Обработчику событий необходим доступ к базе данных

Диспетчер событий предоставляет экземпляр вызывающей сущности и событие

Приводит объект к его фактическому типу Author, чтобы упростить доступ

Перебирает все книги, в которых есть автор, который изменился

Получает авторов в правильном порядке, связанных с этой сущностью Book

Возвращает список имен авторов, но заменяет измененное имя автора на имя, указанное в параметре callingEntity

Создает разделенную запятыми строку с именами авторов книги

Обновляет свойство AuthorsOrdered каждой книги

Возвращение null – быстрый способ сообщить, что обработчик событий всегда успешен

Как видите, обработчик события `Name` намного сложнее и обращается к базе данных несколько раз, что намного медленнее по сравнению

с обработчиками AddReview и RemoveReview. Следовательно, необходимо решить, даст ли кеширование этого значения общий прирост производительности. В данном случае вероятность обновления свойства Name мала, поэтому лучше кешировать список имен авторов для книги.

15.5.3 Добавление свойств в сущность Book с обработкой параллельного доступа

Добавить три свойства – ReviewsCount, ReviewsAverageVotes и AuthorsOrdered – легко. Но проблема может возникнуть, если в одну и ту же сущность Book одновременно (или почти одновременно) добавляются два отзыва. Это может привести к тому, что кешированные значения, связанные с Review, будут неактуальными.

Больше всего времени у меня ушло на обдумывание и разработку оптимального способа обработки одновременных обновлений. Я целыми днями думал обо всех этих вопросах параллелизма, которые могли вызвать проблему, и еще дольше решал, как лучше всего справиться с ними. Эта часть схемы кеширования является наиболее сложной и требует внимательного отношения к ней.

Сначала я подумал об обновлении значений кеша внутри транзакции, но уровень изоляции, необходимый для полностью точного обновления кеша, требовал блокировки большого количества данных. Даже использование чистых команд SQL для вычисления и обновления кеша было небезопасным. (См. увлекательный вопрос и ответ на него на сайте Stack Overflow «Является ли один оператор SQL Server атомарным и последовательным?»: https://stackoverflow.com/q/21468742/1434764.)

Я обнаружил, что лучший способ справиться с проблемой одновременных обновлений – сконфигурировать три значения кеша в качестве маркеров параллелизма (см. раздел 10.6.2). Два одновременных обновления значения вызовут исключение DbUpdateConcurrencyException, которое затем вызывает обработчик параллелизма, написанный для того, чтобы исправить значения кеша на правильные.

На рис. 15.7 показано, что произойдет, если одновременно добавить два отзыва. Это вызовет исключение DbUpdateConcurrencyException. Затем в дело вступает обработчик параллелизма, чтобы исправить значения ReviewsCount и ReviewsAverageVotes.

В этом разделе показаны следующие части обработчика параллелизма:

- код для перехвата любого исключения, созданного методами SaveChanges или SaveChangesAsync;
- обработчик параллелизма верхнего уровня, который находит сущность или сущности Book, вызвавшие исключение DbUpdateConcurrencyException;
- обработчик параллелизма для проблемы с кешированными значениями Review;

- обработчик параллелизма для проблемы с кешированным значением `AuthorsString`.

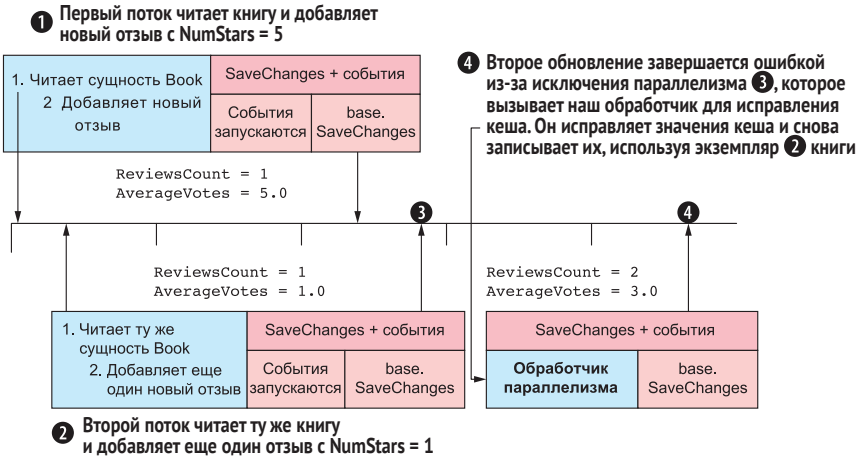


Рис. 15.7 На этом рисунке показано, как два одновременных обновления могут привести к появлению неправильного кешированного значения, что обнаруживается путем настройки свойств `ReviewsCount` и `ReviewsAverageVotes` в качестве маркеров параллелизма. В этом примере будет выброшено исключение `DbUpdateConcurrencyException`, которое будет перехвачено и направлено обработчику параллелизма. Обработчик параллелизма предназначен для обработки этого типа проблемы и исправления значений кеша

КОД ДЛЯ ПЕРЕХВАТА ЛЮБОГО ИСКЛЮЧЕНИЯ, ВЫБРОШЕННОГО МЕТОДАМИ SAVECHANGES ИЛИ SAVECHANGESASYNC

Для перехвата исключения `DbUpdateConcurrencyException` необходимо добавить конструкцию `try/catch` вокруг вызова методов `SaveChanges` или `SaveChangesAsync`. Это дополнение позволяет вызвать обработчик исключений, чтобы попытаться исправить проблему, вызвавшую исключение, или повторно вызвать исключение, если он не сможет решить проблему. Если обработчику удалось корректно обработать исключение, снова вызывается метод `SaveChanges` или `SaveChangesAsync`, чтобы обновить базу данных.

В этом конкретном случае нужно рассмотреть еще одну проблему: пока мы исправляли первое обновление, могло произойти еще одно. Конечно, такой сценарий маловероятен, но и с ним нужно разобраться; в противном случае второй вызов метода `SaveChanges` или `SaveChangesAsync` завершится ошибкой. По этой причине нам понадобится внешний цикл языка C# `do/while`, чтобы продолжать повторять вызов методов `SaveChanges` или `SaveChangesAsync` до тех пор, пока он не будет успешным или не возникнет исключение, которое невозможно исправить.

Кроме того, библиотека `GenericEventRunner` позволяет зарегистрировать обработчик исключений, который будет вызываться, если

вышеуказанные методы выбросят исключение. Ваш обработчик исключений должен вернуть `IStatusGeneric`, и есть три возможных варианта:

- *в статусе ошибок нет*. Обработчик исключений устранил проблему, и для обновления базы данных необходимо снова вызвать методы `SaveChanges` или `SaveChangesAsync`;
- *статус возвращает ошибки*. Обработчик исключений преобразовал исключение в сообщение (сообщения) об ошибках. Этот подход полезен для превращения исключений базы данных в удобные для пользователя сообщения об ошибках;
- *статус возвращает* `null`. Обработчик исключений не может обработать исключение, и оно должно быть выброшено повторно.

Листинг 15.8 Упрощенная версия вызова метода SaveChanges с GenericEventRunner

```
private IStatusGeneric<int>
    CallSaveChangesWithExceptionHandler
    (DbContext context,
        Func<int> callBaseSaveChanges)
{
    var status = new StatusGenericHandler<int>();
    do
    {
        try
        {
            int numUpdated = callBaseSaveChanges();
            status.SetResult(numUpdated);
            break;
        }
        catch (Exception e)
        {
            IStatusGeneric handlerStatus
                = … ВАШ КОД ОБРАБОТКИ ИСКЛЮЧЕНИЯ;
            if (handlerStatus == null)
                throw;
            status.CombineStatuses(handlerStatus);
        }
    } while (status.IsValid);

    return status;
}
```

Возвращаемое значение представляет собой статус с int, возвращенным из метода SaveChanges

Базовый метод SaveChanges предоставляется для вызова из обработчика

Статус, который будет возвращен

Вызов метода SaveChanges выполняется в пределах цикла do/while

Вызывает базовый метод SaveChanges

Если исключения нет, устанавливает статус результата и выходит из цикла do/while

Перехват кеширует любые исключения, выбрасываемые методом SaveChanges

Здесь вызывается обработчик исключения, который возвращает значение null или статус

Если обработчик исключения возвращает null, исходное исключение генерируется повторно ...

... в противном случае любые ошибки из обработчика исключений добавляются к основному статусу

Возвращает статус

Если обработчик исключений был успешным, он повторяет цикл, чтобы попытаться снова вызвать метод SaveChanges

ОБРАБОТЧИК ВЕРХНЕГО УРОВНЯ, ОБНАРУЖИВАЮЩИЙ СУЩНОСТЬ (СУЩНОСТИ) BOOK, ВЫЗЫВАЮЩУЮ(ИЕ) ИСКЛЮЧЕНИЕ

Решение проблемы параллелизма включает в себя несколько общих частей, поэтому мы создаем обработчик параллельного доступа для

управления этими частями. В следующем листинге показан метод обработчика параллельного доступа верхнего уровня `HandleCacheValuesConcurrency`.

Листинг 15.9 Обработчик параллельного доступа верхнего уровня, содержащий общий код исключения

```
public static IStatusGeneric HandleCacheValuesConcurrency
    (this Exception ex, DbContext context)
{
    var dbUpdateEx = ex as DbUpdateConcurrencyException;
    if (dbUpdateEx == null)
        return null;

    foreach (var entry in dbUpdateEx.Entries)
    {
        if (!(entry.Entity is Book bookBeingWrittenOut))
            return null;

        var bookThatCausedConcurrency = context.Set<Book>()
            .IgnoreQueryFilters()
            .AsNoTracking()
            .SingleOrDefault(p => p.BookId
                == bookBeingWrittenOut.BookId);

        if (bookThatCausedConcurrency == null)
        {
            entry.State = EntityState.Detached;
            continue;
        }

        var handler = new FixConcurrencyMethods(entry, context);
        handler.CheckFixReviewCacheValues(
            bookThatCausedConcurrency, bookBeingWrittenOut);

        handler.CheckFixAuthorOrdered(
            bookThatCausedConcurrency, bookBeingWrittenOut);
    }

    return new StatusGenericHandler();
}
```

Если исключение не типа DbUpdateConcurrencyException, возвращаем null, чтобы сообщить, что мы не можем обработать это исключение

Этот метод расширения решает проблемы параллелизма с кешированными значениями Review и Author

Приводит исключение к DbUpdateConcurrencyException

Должна быть только одна сущность, но в случае массовой загрузки обрабатывается много сущностей

Приводит сущность к типу Book. Если это не Book, возвращаем null, чтобы сообщить, что метод не может ее обработать

Если ни одной книги не было найдено, значит, она была удалена, текущая книга помечается как отсоединенная, чтобы она не обновлялась

Читает неотслеживаемую версию Book из базы данных. (Обратите внимание на IgnoreQueryFilters. Возможно, она была мягко удалена)

Создает класс, содержащий кешированные значения Review и AuthorsOrdered

Устраняет любые проблемы параллелизма с кешированными значениями Review

Устраняет любые проблемы параллелизма с кешированным значением AuthorsOrdered

Возвращает действительный статус, чтобы сообщить, что проблема параллелизма была устранена

ОБРАБОТЧИК ПАРАЛЛЕЛЬНОГО ДОСТУПА ДЛЯ ПРОБЛЕМЫ С КЕШИРОВАННЫМИ ЗНАЧЕНИЯМИ REVIEW

Обработчик параллельного доступа `CheckFixReviewCacheValues` имеет дело только с кешированными значениями `Review`. Его задача – объединить кешированные значения `Review` в записываемой сущности

и кешированные значения `Review`, которые были добавлены в базу данных. В этом методе используется тот же стиль дельта-обновления, что и в обработчике событий кешированных значений `Review`. В следующем листинге показан обработчик параллельного доступа `CheckFixReviewCacheValues`.

ПРИМЕЧАНИЕ Если вы незнакомы с обработкой параллельного доступа в EF Core, рекомендую обратиться к разделу 10.6.3, где описаны различные типы данных, участвующие в обработке исключения параллелизма.

Листинг 15.10 Код для исправления одновременного обновления кешированных значений `Review`

Этот метод обрабатывает ошибки параллельного доступа в кешированных значениях Review

Этот параметр представляет собой книгу из базы данных, которая вызвала проблему параллельного доступа

```
public void CheckFixReviewCacheValues(
    Book bookThatCausedConcurrency,
    Book bookBeingWrittenOut)
{
    var previousCount = (int)_entry
        .Property(nameof(Book.ReviewsCount))
        .OriginalValue;
    var previousAverageVotes = (double)_entry
        .Property(nameof(Book.ReviewsAverageVotes))
        .OriginalValue;

    if (previousCount ==
        bookThatCausedConcurrency.ReviewsCount
        && previousAverageVotes ==
           bookThatCausedConcurrency.ReviewsAverageVotes)
        return;

    var previousTotalStars = Math.Round(
        previousAverageVotes * previousCount);

    var countChange =
        bookBeingWrittenOut.ReviewsCount
      - previousCount;
    var starsChange = Math.Round(
          bookBeingWrittenOut.ReviewsAverageVotes
          * bookBeingWrittenOut.ReviewsCount)
     - previousTotalStars;
    var newCount =
        bookThatCausedConcurrency.ReviewsCount
        + countChange;
    var newTotalStars = Math.Round(
            bookThatCausedConcurrency.ReviewsAverageVotes
            * bookThatCausedConcurrency.ReviewsCount)
        + starsChange;
```

Этот параметр – книга, которую вы пытались обновить

Сохраняет количество и оценку в базе данных до того, как события изменили их

Вычисляет оценку перед применением нового обновления

Если предыдущее количество и оценка совпадают с текущими из базы данных, проблема параллельного доступа отсутствует, поэтому метод возвращается

Обрабатывает комбинированное изменение из текущей книги и других обновлений, внесенных в базу данных

Получает изменение, которое событие пытается внести в кешированные значения

```
    _entry.Property(nameof(Book.ReviewsCount))
        .CurrentValue = newCount;
    _entry.Property(nameof(Book.ReviewsAverageVotes))
        .CurrentValue = newCount == 0
        ? 0 : newTotalStars / newCount;

    _entry.Property(nameof(Book.ReviewsCount))
        .OriginalValue = bookThatCausedConcurrency
        .ReviewsCount;
    _entry.Property(nameof(Book.ReviewsAverageVotes))
            .OriginalValue =
        bookThatCausedConcurrency
            .ReviewsAverageVotes;
}
```

Настраивает кешированные значения Review с пересчитанными значениями

Настраивает OriginalValue для кешированных значений Review в текущую базу данных

Да, это довольно сложный код, поэтому я даю переменным «говорящие» имена. Даже я могу потеряться здесь, если вернусь к нему через несколько месяцев.

ОБРАБОТЧИК ПАРАЛЛЕЛЬНОГО ДОСТУПА ДЛЯ ПРОБЛЕМЫ С КЕШИРОВАННЫМ ЗНАЧЕНИЕМ AUTHORSSTRING

У обработчика параллельного доступа `CheckFixAuthorsOrdered` тот же формат, что и у метода `CheckFixReviewCacheValues`, но он работает с кешированным значением `AuthorsOrdered`. Его задача – сопоставить кешированное значение `AuthorsOrdered` в записываемой сущности и кешированное значение `AuthorsOrdered`, которое было добавлено в базу данных. В результате обработчик параллелизма `CheckFixAuthorsOrdered`, показанный в следующем листинге, должен использовать стиль повторного вычисления обновления, поскольку здесь нельзя использовать подход с дельта-обновлением.

Листинг 15.11 Код для исправления одновременного обновления кешированного значения `AuthorsOrdered`

Этот метод обрабатывает ошибки параллельного доступа в кешированном значении AuthorsOrdered

Этот параметр представляет собой книгу из базы данных, которая вызвала проблему параллелизма

```
public void CheckFixAuthorsOrdered(
    Book bookThatCausedConcurrency,
    Book bookBeingWrittenOut)
{
    var previousAuthorsOrdered = (string)_entry
        .Property(nameof(Book.AuthorsOrdered))
        .OriginalValue;

    if (previousAuthorsOrdered ==
        bookThatCausedConcurrency.AuthorsOrdered)
        return;
```

Этот параметр – книга, которую вы пытались обновить

Получает предыдущую строку AuthorsOrdered до того, как событие обновило ее

Если предыдущее значение AuthorsOrdered соответствует текущему AuthorsOrdered из базы данных, проблема с параллелизмом AuthorsOrdered отсутствует, поэтому метод возвращает управление

```
        var allAuthorsIdsInOrder = _context.Set<Book>()
            .IgnoreQueryFilters()
            .Where(x => x.BookId ==
                        bookBeingWrittenOut.BookId)
            .Select(x => x.AuthorsLink
                .OrderBy(y => y.Order)
                .Select(y => y.AuthorId)).ToList()
            .Single();

        var namesInOrder = allAuthorsIdsInOrder
            .Select(x => _context.Find<Author>(x).Name);

        var newAuthorsOrdered =
            string.Join(", ", namesInOrder);

        _entry.Property(nameof(Book.AuthorsOrdered))
            .CurrentValue = newAuthorsOrdered;

        _entry.Property(nameof(Book.AuthorsOrdered))
            .OriginalValue =
            bookThatCausedConcurrency.AuthorsOrdered;
    }
```

Получает AuthorId для каждого автора, связанного с этой книгой, в правильном порядке

Получает имя каждого автора с помощью метода Find

Создает список авторов, разделенных запятыми

Устанавливает OriginalValue в текущее кешированное значение AuthorsOrdered из базы данных

Отсюда можно установить кешированное значение AuthorsOrdered с комбинированными значениями

Важно отметить, что нужно получать классы сущностей `Author`, используя метод `Find`, потому что сущность `Author`, создавшая обновление для кешированного значения `AuthorsOrdered`, еще не была записана в базу данных. `Find` – единственный метод, который сначала проверяет DbContext текущего приложения на предмет отслеживаемых сущностей, чтобы найти нужную сущность. Он загружает отслеживаемую сущность с этим `AuthorId`, вместо того чтобы загружать из базы данных версию, которая еще не была обновлена.

15.5.4 Добавление системы проверки и восстановления в систему событий

Со времени первого издания я настроил несколько клиентских систем и создал систему кеширования, которая охватывает все возможные варианты. Но, возможно, я что-то пропустил, поэтому добавил отдельную систему проверки и восстановления, которая будет работать вместе с моей системой кеширования, чтобы сообщать мне о наличии проблемы. Эта система позволяет мне спать по ночам, а моим клиентам нравится, что они могут быть уверены, что их данные актуальны.

Возможно, вы подумаете, что такой подход является излишним, но если вы добавляете систему кеширования в существующую систему, вам все равно понадобится способ заполнить кешированные значения для существующих данных. Обычно я создаю код для добавления кешированных значений в рабочую базу данных текущего приложения перед выпуском новой версии приложения, использующей кешированные значения в запросе. Требуется лишь немного больше уси-

лий, чтобы превратить этот код в полезный сервис, который можно использовать для проверки и исправления кешированных значений.

В качестве примера я встроил систему проверки и восстановления в приложение Book App. Этот сервис под названием `CheckFixCacheValuesService` доступен в приложении ASP.NET Core. Его можно использовать при проверке и восстановлении по мере необходимости. Вместо того чтобы подробно описывать код, я привел рис. 15.8, где показан обзор того, что делает класс `CheckFixCacheValuesService`.

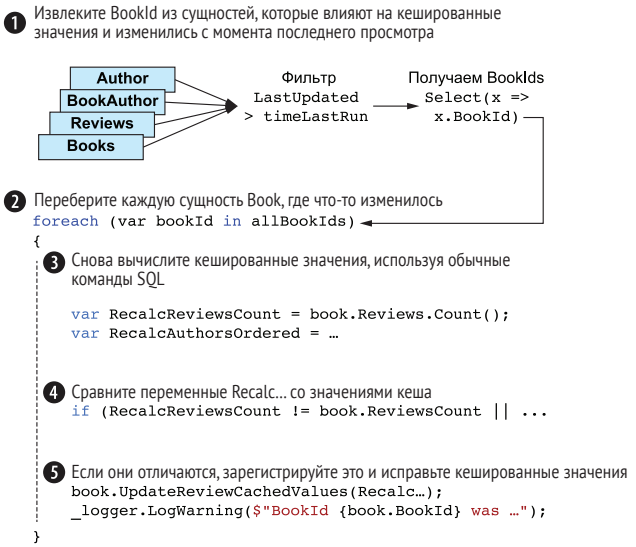


Рис. 15.8 Пять этапов `CheckFix` в версии приложения Book App для третьей части. Этот код запускается из фоновой службы, которая периодически проверяет базу данных на предмет наличия сущностей, которые изменились и потенциально могли изменить кешированные значения. Поскольку в этом коде используется другой способ поиска и вычисления кешированных значений, он найдет любые кешированные значения, которые не являются актуальными, и исправит их

> **ПРИМЕЧАНИЕ** Класс `CheckFixCacheValuesService` и связанные с ним классы находятся в репозитории GitHub для этой книги в папке под названием CheckFixCode в проекте BookApp.Infrastructure.Books.EventHandlers. Кроме того, можно найти фоновую службу в проекте BookApp.BackgroundTasks.

Недостаток кода, показанного на рис. 15.8, в том, что он добавляет дополнительные обращения к базе данных, а это может повлиять на производительность системы. В приложении Book App, например, обновление класса сущности вызывает обновление свойства `LastUpdatedUtc` (см. раздел 11.4.3). Код проверки и восстановления может

довольно быстро найти все сущности, которые были изменены, скажем, за последние 24 часа (в тестовой базе данных 700 000 сущностей, и сканирование занимает всего около 10 мс), но каждая проверка измененной сущности занимает 5 мс. Так что если ваше приложение обрабатывает множество изменений за день, то на выполнение такого кода уйдет время.

По этой причине подобного рода система проверки и восстановления запускается в то время, когда в системе не так много пользователей – ночью, в выходные дни – или вручную администратором, когда он подозревает проблему. Система вряд ли что-нибудь найдет, но если она обнаружит неверно заданное значение, то вы будете знать, что в вашем коде есть ошибка.

В версии приложения Book App для третьей части есть примеры ночного и ручного запусков системы проверки и восстановления. Фоновая служба ASP.NET Core запускает `CheckFixCacheValuesService` в 1:00 каждое утро (был часовой пояс по Гринвичу, но из-за сбоя в Linux теперь используется UTC), и вы можете запустить службу проверки и восстановления вручную, выбрав пункт меню **Admin** > **Check Cached Vals**.

> **ПРЕДУПРЕЖДЕНИЕ** Реализация службы `CheckFixCacheValuesService` в приложении Book App предполагает, что обновления базы данных не происходят, когда он исправляет неправильные значения. При возникновении исключений параллелизма `CheckFixCacheValuesService` потребуется собственный обработчик исключений параллелизма.

15.6 Сравнение четырех подходов к производительности с усилиями по разработке

В начале этой главы я сравнил прирост производительности четырех подходов. Хотя улучшения производительности неоспоримы, есть и другие факторы, которые следует учитывать при рассмотрении каждого подхода к настройке производительности, например сколько усилий по разработке потребовалось для каждого подхода, понадобились ли для них конкретные навыки и насколько сложными были эти решения.

В данном разделе я рассмотрю эти соображения и предоставлю больше информации, чтобы попытаться ответить на некоторые вопросы, касающиеся разработки. На рис. 15.9 показано краткое изложение четырех способов улучшить производительность приложения, требуемые навыки и время разработки.

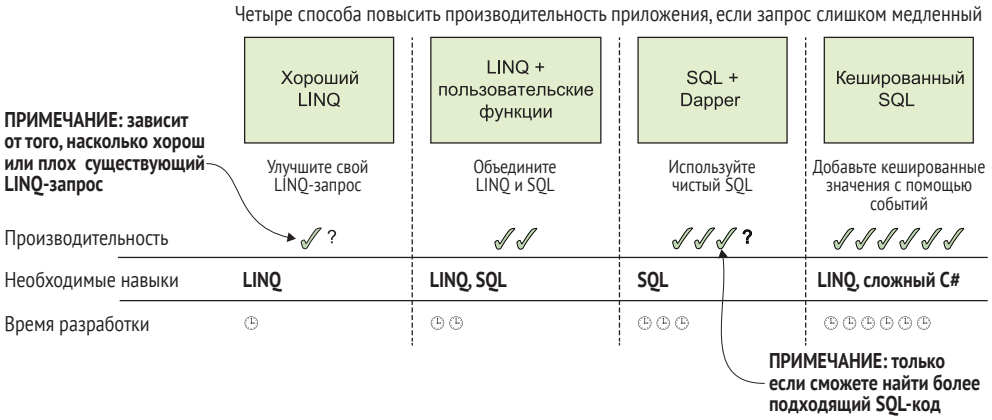


Рис. 15.9 Четыре подхода к повышению производительности запроса. Каждый подход оценивается в плане повышения производительности (больше галочек означает лучшую производительность). Здесь показаны навыки, необходимые, чтобы применить этот подход, и время разработки, необходимое для реализации кода, связанного с этим подходом

В табл. 15.2 представлено текстовое изложение четырех подходов с точки зрения необходимых усилий и навыков.

В целом я доволен процессом. EF Core генерирует отличный код SQL с самого начала, но только в случае, если ваши LINQ-запросы написаны в соответствии с принципами работы EF Core. Методы и подходы, представленные в первых шести главах, – хорошая отправная точка для написания хороших запросов LINQ.

Как было сказано в главе 14, убедитесь, что ваши стандартные шаблоны запросов работают хорошо; в противном случае вы добавите проблемы с производительностью в свое приложение с самого начала. Но в какой-то момент нужно будет настроить производительность приложения EF Core; в этой главе представлено множество идей и подходов, которые могут помочь.

Таблица 15.2 Объем усилий, необходимых для применения четырех подходов к приложению Book App

Подход	Усилие + навыки	Комментарии
Хороший LINQ	Время: низкое (создано в главе 2) Навыки: LINQ, DDD	Выборочный запрос – это тот же запрос, который я использовал в главе 2, и он хорошо работает. Ключевой частью была разработка того, как получить среднее значение свойства `NumStar` сущности `Review` в базе данных (см. раздел 6.1.8). Кроме того, изменение класса сущности в DDD-стиле означало, что цена была доступна как отдельное свойство, к которому можно было добавить индекс SQL
LINQ + пользовательские функции	Полдня LINQ + SQL	Я обнаружил, что пользовательские функции (см. раздел 10.1) – хороший способ сохранить подход с LINQ, но заменить часть LINQ-запроса, которая работает плохо, не так просто, как хотелось бы. Пользовательские функции полезны только в том случае, если вы можете найти более подходящий SQL-код, чтобы поместить его в них

Таблица 15.2 (*окончание*)

Подход	Усилие + навыки	Комментарии
SQL + Dapper	Полдня на изучение, полдня на написание SQL	Этот подход требовал изучения SQL, генерируемого EF Core, и выяснения, могу ли я что-нибудь сделать для его улучшения. Я нашел только одну оптимизацию (сортировка по оценкам), но эта функция является ключевой. Переписывать SQL-код так, чтобы в нем были все фильтры, сортировки и разбиение по страницам, было немного утомительно – намного сложнее, чем использовать LINQ
LINQ + кеширование	LINQ + кеширование: около недели, но в следующий раз будет быстрее Проверка/ восстановление: полтора дня Сложный C#, параллелизм	Этот подход – безусловно, тяжелая работа, но он дает фантастический результат. Потребовалось много времени, чтобы выработать лучший способ обработки одновременных обновлений и тестирования, но, реализовав этот подход однажды, в следующий раз вы справитесь быстрее. Код проверки и восстановления занял немного больше времени, но, как я уже сказал, мне все равно пришлось бы писать его, если бы я оптимизировал производительность существующего приложения, в котором уже были пользовательские данные. Еще одним источником времени, потраченного на внедрение системы кеширования, была работа, которую я выполнял для клиента. На создание единой системы дельта-кеширования значений у меня ушло 11 часов, но мне не нужно было выполнять обработку параллельного доступа, поскольку клиентское приложение останавливало все повторяющиеся обновления данных пользователем

15.7 Повышение масштабируемости базы данных

Все четыре подхода к настройке производительности связаны со скоростью: как быстро вернуть результат пользователю. Но есть еще один аспект, который следует учитывать, – это *масштабируемость*: обработка большого количества пользователей, заходящих на сайт одновременно. В заключение этой главы рассмотрим масштабируемость базы данных. В разделе 14.7 говорится о масштабируемости базы данных с точки зрения возможности приобретения более мощного оборудования для запуска сервера базы данных, поскольку эта книга посвящена EF Core. Но важнее всего общая масштабируемость приложения. По этой причине я всегда показываю производительность всего приложения, потому что именно это и увидит конечный пользователь. Если сосредоточить внимание на общей производительности приложения, то это позволит не тратить много времени на то, чтобы сократить продолжительность доступа к базе данных на несколько миллисекунд, когда у клиентской части уходит более 100 мс для отображения данных.

Первое, что следует сделать для улучшения масштабируемости, – использовать асинхронный доступ к базе данных. Асинхронные команды, используемые в приложении ASP.NET Core, высвобождают поток, который может использоваться другим кодом, тем самым

спасая пул потоков ASP.NET Core от истощения (см. раздел 5.10.1). У асинхронных команд есть небольшой недостаток – они выполняются немного дольше (подробные сведения см. в разделе 14.7.2), но в целом асинхронный режим подходит для любого приложения, с которым одновременно работает много пользователей. Версия приложения Book App для третьей части использует асинхронные команды повсюду.

Другие полезные изменения, которые можно внести с помощью таких приложений, как ASP.NET Core, – это запуск более мощных экземпляров приложения (известно как *вертикальное масштабирование*) и запуск дополнительных экземпляров приложения (т. н. *горизонтальное масштабирование*). Вы также можете заплатить за более мощное оборудование для работы вашего сервера базы данных.

> **ПРИМЕЧАНИЕ** Все подходы, использованные в этой главе, будут работать в приложении с применением нескольких экземпляров ASP.NET Core, включая подход *LINQ + кеширование*. Однако ночная служба проверки и восстановления должна запускаться в отдельном WebJob, а не в качестве фоновой службы ASP.NET Core.

Один из основных фактов, касающихся масштабируемости базы данных, заключается в том, что чем быстрее вы выполняете запросы к базе данных, тем большее количество одновременных обращений она может обрабатывать. Уменьшение количества обращений к базе данных также снижает нагрузку на нее (см. раздел 14.5.1). К счастью, начиная с EF Core 3 тип запроса по умолчанию загружал любые коллекции в рамках одного обращения к базе данных. Кроме того, отложенная загрузка может показаться отличной экономией времени, но она добавляет обратно все эти отдельные обращения к базе данных, и масштабируемость и производительность могут пострадать.

У некоторых больших приложений бывает много одновременных обращений к базе данных, и нужен выход из этой ситуации. Первый и самый простой подход – заплатить за более мощную базу данных. Если это решение не поможет, вот несколько идей, которые стоит рассмотреть:

- разделите данные по нескольким базам данных: используйте шардинг.
  Если ваши данные каким-либо образом разделены (например, если у вас финансовое приложение, которое используют многие малые предприятия), то можно распределить данные каждой компании по разным базам данных, т. е. по одной базе данных для каждой компании. Такой подход называется шардингом (http://mng.bz/veN4). В разделе 11.7 показан простой способ реализации с помощью EF Core;
- разделите вашу базу данных на чтение и запись: архитектура CQRS.

Архитектура CQRS (https://martinfowler.com/bliki/CQRS.html) отделяет операции чтения от операций записи. Этот подход позволяет оптимизировать чтение и, возможно, использовать отдельную базу данных или несколько баз данных только для операций чтения CQRS;

- смешивайте реляционные и нереляционные базы данных: polyglot persistence.
  Подход с кешированным SQL делает сущность `Book` похожей на полное определение книги, которую будет содержать структура JSON. Применяя архитектуру CQRS, можно было бы использовать реляционную базу данных для обработки любых записей, но при любой записи вы могли бы создать версию книги в формате JSON и записывать ее в нереляционную базу данных для чтения или в несколько баз данных. Этот подход, обеспечивающий более высокую производительность чтения, является одной из форм polyglot persistence (http://mng.bz/K4RX). В разделе 16.3 мы реализуем смешанное приложение с реляционной и нереляционной базами данных, чтобы получить еще бóльшую производительность, особенно с точки зрения масштабируемости.

Резюме

- Если вы тщательно создаете LINQ-запросы и пользуетесь всеми их возможностями, то EF Core вознаградит вас созданием отличного SQL-кода.
- Можно использовать функциональность EF Core `DbFunction`, чтобы внедрить фрагмент кода SQL, содержащийся в пользовательской функции, в LINQ-запрос. Эта функция позволяет настраивать часть запроса EF Core, выполняемого на сервере базы данных.
- Если запрос к базе данных выполняется медленно, проверьте код SQL, создаваемый EF Core. Его можно получить, просмотрев сообщения с уровнем журналирования `Information`, создаваемые EF Core.
- Если вы чувствуете, что можете создать более подходящий SQL-код для запроса, чем EF Core, то можно использовать несколько методов для вызова SQL из EF Core или Dapper для непосредственного выполнения SQL-запроса.
- Если все другие подходы к настройке производительности не обеспечивают требуемой производительности, рассмотрите возможность изменения структуры базы данных, включая добавление свойств для хранения кешированных значений. Но будьте осторожны.
- Помимо сокращения времени, затрачиваемого на запрос, следует учитывать масштабируемость приложения, т. е. поддержку множества одновременных пользователей. Во многих приложениях,

таких как ASP.NET Core, использование асинхронных команд EF Core может улучшить масштабируемость.

- Глава 16 предлагает еще один способ улучшить масштабируемость и производительность, добавив в приложение Book App базу данных Cosmos DB.

Для читателей, знакомых с EF6:

- в EF6.x нет функциональности `DbFunction`, которая так упрощает вызов пользовательской функции в EF Core.

16 Cosmos DB, CQRS и другие типы баз данных

В этой главе рассматриваются следующие темы:

- нереляционные (NoSQL) базы данных и чем они отличаются от реляционных;
- изучение возможностей NoSQL базы данных Cosmos DB;
- настройка производительности приложения Book App с помощью провайдера базы данных Cosmos DB;
- рассмотрение различий и ограничений использования Cosmos DB с EF Core 5;
- проблемы, с которыми вы можете столкнуться при переходе с одного типа базы данных на другой.

Приложение Book App было постоянной темой в этой книге, и до сих пор оно использовало базу данных SQL Server для хранения данных о книгах. В этой главе мы улучшим производительность приложения, используя исходную базу данных SQL Server совместно с базой данных Cosmos DB. В главе 14 мы настроили производительность нашего приложения для обработки 100 000 книг. В этой главе число книг будет доведено до 500 000 с такой же или более высокой производительностью с помощью Cosmos DB. Это относительно новая база

данных (она появилась в 2017 году), и, возможно, некоторые читатели ее пока не использовали. Поэтому, помимо использования этой базы данных для повышения производительности и масштабируемости, я указываю на различия между Cosmos DB, которая представляет собой NoSQL базу данных, и более традиционными реляционными базами данных, такими как SQL Server.

Cosmos DB и реляционные базы данных сильно различаются, но, кроме того, существуют небольшие различия и между реляционными базами данных, поддерживаемыми EF Core. В конце главы приводится список вещей, которые нужно проверить и изменить, если вы переходите с одного типа реляционной базы данных на другой.

16.1 Различия между реляционными и нереляционными базами данных

ЧТОБЫ СЭКОНОМИТЬ ВРЕМЯ Пропустите этот раздел, если вы уже знакомы с NoSQL базами данных.

Cosmos DB не похожа на базы данных, о которых уже шла речь в этой книге, например SQL Server, PostgreSQL и SQLite. Cosmos DB – это NoSQL база данных, тогда как SQL Server, PostgreSQL и SQLite (наряду со многими другими) – это реляционные базы данных.

Как вы уже знаете, реляционные базы данных используют первичные и внешние ключи для формирования связей между таблицами, которые EF Core превращает в навигационные свойства. Реляционные базы данных превосходно справляются со связями со множеством правил (которые называются ограничениями), чтобы убедиться, что эти связи (relationships) соответствуют дизайну, который вы выбрали для своей базы данных. Поэтому они и называются реляционными (relational) базами данных.

Реляционные базы данных существуют уже несколько десятилетий, и почти все они используют язык SQL, а это означает, что каждая реализация реляционной базы данных похожа на все остальные. Поэтому переключиться, скажем, с SQL Server на PostgreSQL не так уж сложно, особенно если вы используете EF Core, который скрывает некоторые различия. Долгая история реляционных баз данных также означает, что можно найти множество их реализаций, множество инструментов и помощь при работе с этими базами.

С другой стороны, NoSQL базы данных спроектированы таким образом, чтобы обеспечивать высокую производительность с точки зрения скорости, масштабируемости и *доступности* (возможность переключения на другую базу данных в случае сбоя одной из них). Здесь нет общего языка, такого как SQL, поэтому каждая реализация идет своим путем, чтобы максимизировать функции, на которых она хочет сосредоточиться. Для достижения этих целей производитель-

ности NoSQL базы данных отказываются от некоторых правил, применяемых реляционными базами данных.

Например, многие NoSQL базы данных позволяют использовать несколько экземпляров одной и той же базы данных для обеспечения масштабируемости и доступности. Для этого в таких базах отсутствует реляционное правило, согласно которому данные всегда согласованы, т. е. вы всегда будете получать самые свежие данные. NoSQL базы данных являются *согласованными в конечном счете*. Это означает, что обновление одного экземпляра базы данных может занять некоторое время (в идеале секунды или меньше), прежде чем начнется обновление другого.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ Если вы хотите изучить типы и различия между реляционными и NoSQL базами данных, рекомендую статью Microsoft: http://mng.bz/9Nzj.

16.2 Cosmos DB и ее провайдер для EF Core

Как уже было сказано, Cosmos DB не следует принципу работы реляционных баз данных. Конечно, у нее есть база данных и даже несколько псевдо-SQL-команд, но в остальном она сильно отличается от реляционных баз данных. Однако поддержка Cosmos DB со стороны EF Core обеспечивает общий интерфейс, упрощающий работу с Cosmos DB для тех, кто уже знает EF Core.

В этой главе мы рассмотрим возможности как самой Cosmos DB, так и ее текущего провайдера базы данных для EF Core. Следует отметить: я говорю «текущий провайдер базы данных Cosmos DB», потому что провайдер базы данных еще далек от завершения, о чем я подробно расскажу в этой главе.

Чтобы понять, почему провайдер базы данных Cosmos DB не был улучшен, достаточно взглянуть на статистику: количество загрузок Cosmos DB составляет всего 1 % от количества загрузок SQL Server. Команда EF Core руководствуется тем, что нужно разработчикам, и, будучи небольшой командой, она не может делать все. Поэтому провайдер базы данных Cosmos DB не был улучшен в EF Core 5. Но, как вы увидите далее, я успешно использовал провайдера базы данных EF Core 5 Cosmos DB для повышения производительности приложения Book App.

Итак, почему я посвятил эту главу Cosmos DB, если у ее провайдера есть ограничения, и зачем ее читать? Для некоторых приложений использование NoSQL баз данных обеспечивает лучшую производительность и масштабируемость, по сравнению с реляционной базой данных за ту же цену. Кроме того, в дорожной карте для EF Core 6 (http://mng.bz/Wreg) есть раздел об улучшении поддержки EF Core для Cosmos DB, поэтому я надеюсь, что некоторые (если не многие) ограничения, описанные в этой главе, будут сняты.

Поскольку текущий провайдер базы данных Cosmos DB, вероятно, улучшится, я тщательно отделяю различия между Cosmos DB и реляционной базой данных и ограничения провайдера базы данных. Это гарантирует, что глава будет по-прежнему полезна, когда будут выпущены улучшенные версии провайдера Cosmos DB.

ПРИМЕЧАНИЕ Это сравнение реляционных и NoSQL баз данных *не* говорит о том, что одни лучше других; у обеих концепций есть свои сильные и слабые стороны. Кроме того, Cosmos DB является одной из реализаций NoSQL базы данных, поэтому ее ограничения будут отличаться из других реализаций NoSQL баз данных. Это сравнение приводится здесь для того, чтобы выделить части Cosmos DB, которые работают иначе по сравнению с реляционными базами данных, существующими уже много лет.

Другая причина рассмотреть различия между Cosmos DB и реляционными базами данных – дать ряд рекомендаций по поводу того, когда можно использовать Cosmos DB вместо реляционной базы данных. В разделе 16.6.1 описывается много различий между Cosmos DB и реляционными базами данных, а также приводятся отличия, отмеченные примечаниями, которые начинаются словами ОТЛИЧИЕ В COSMOS DB, см. следующий пример.

ОТЛИЧИЕ В COSMOS DB Эта функция в Cosmos DB работает не так, как в реляционных базах данных.

Другая область, которую я хочу выделить, – это ограничения провайдера базы данных EF Core 5 для Cosmos DB. Это области, для которых в EF Core 5 не реализован код для использования всех функций Cosmos DB (но имейте в виду, что в будущих версиях EF Core некоторые из этих ограничений могут стать неактуальными). В разделе 16.6.3 рассказывается о множестве ограничений поставщика базы данных EF Core 5 для Cosmos DB и указаны и другие ограничения, обозначенные примечаниями, которые начинаются со слов ОГРАНИЧЕНИЕ EF CORE 5; см. следующий пример.

ОГРАНИЧЕНИЕ EF CORE 5 Это ограничение применяется к текущему провайдеру базы данных EF Core 5 Cosmos DB.

16.3 Создание системы CQRS с использованием Cosmos DB

Чтобы получить правильное представление о Cosmos DB, нужно создать что-то реальное. В разделе 15.7 я предположил, что архитектура CQRS может обеспечить большую масштабируемость. Добавление

системы CQRS, использующей Cosmos DB, – нетривиальный процесс, поэтому этот пример продемонстрирует множество различий между Cosmos DB и реляционными базами данных. Я также надеюсь, что он предоставит вам еще один метод, который вы можете использовать для оптимизации производительности собственных приложений.

В этом разделе мы реализуем архитектуру CQRS, используя структуру базы данных «полиглот», которая обеспечит лучшую производительность и масштабируемость.

ОПРЕДЕЛЕНИЕ Архитектура CQRS отделяет операции чтения от операций, которые обновляют данные, используя раздельные интерфейсы. Эта архитектура может максимизировать производительность, масштабируемость и безопасность, а также облегчить развитие системы с течением времени за счет большей гибкости. См. http://mng.bz/Ix8D.

ОПРЕДЕЛЕНИЕ Структура базы данных «полиглот» использует комбинацию типов хранилищ: реляционные базы данных, NoSQL базы данных, плоские файлы и т. д. Идея состоит в том, что у каждого типа базы данных есть сильные и слабые стороны, и, используя две или более базы данных, можно получить лучшую систему в целом. См. http://mng.bz/6r1W.

Архитектура CQRS основана на том, что в приложении чтение данных отличается от записи. Чтение часто бывает сложным, поскольку данные извлекаются из нескольких мест, тогда как во многих приложениях (но не во всех) запись может быть более простой и менее обременительной. В текущем приложении Book App видно, что перечисление книг – непростая задача, но добавить отзыв довольно просто. Разделение кода для каждой части может помочь вам сосредоточиться на конкретных функциях каждой части – еще одно применение принципа разделения ответственностей.

В главе 15 мы создали версию приложения, в которой кешировали значения (см. раздел 15.5). Тогда я понял, что окончательный запрос не имеет доступа к каким-либо связям и его можно сохранить в более простой базе данных, например NoSQL. В этом примере мы будем использовать структуру базы данных «полиглот» со смесью реляционных и NoSQL баз данных по следующим причинам:

- использование реляционной базы данных с возможностью записи не лишено смысла, поскольку бизнес-приложения часто используют реляционные данные. Представьте себе настоящий сайт по продаже книг, в котором много сложных, связанных данных для обработки таких аспектов бизнес-логики, как поставщики, запасы, цены, заказы, оплата, доставка, отслеживание и аудит. Я думаю, что хорошо известная реляционная база данных с ее превосходным уровнем целостности данных была бы хорошим выбором для решения многих бизнес-задач;

- но отношения и некоторые аспекты реляционной базы данных, например необходимость динамического вычисления некоторых значений, могут замедлить получение данных. Таким образом, NoSQL база данных с предварительно вычисленными значениями, например средняя оценка книги, может значительно улучшить производительность по сравнению с реляционной базой данных. Проекция CQRS со стороны чтения – это то, что Матеуш Сташ называет «легитимным кешем» в своей статье: http://mng.bz/A7eC.

На основе этих исходных данных мы разработаем то, что я называю двухбазовой архитектурой CQRS, как показано на рис. 16.1.

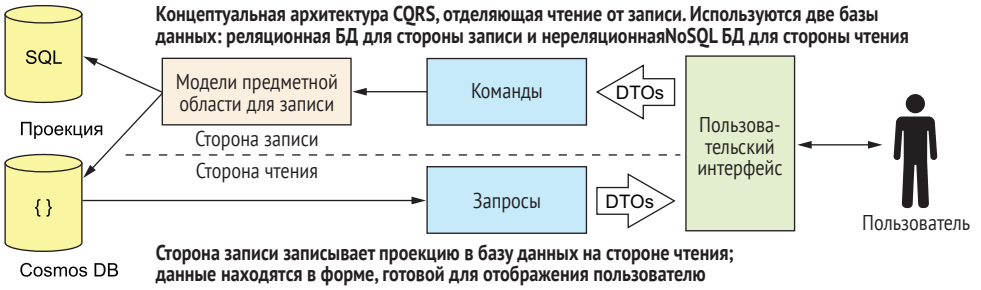


Рис. 16.1 Концептуальное представление архитектуры CQRS: реляционная база данных для записи и NoSQL база данных для чтения. Запись требует немного больше работы, потому что запись ведется в две базы данных: обычную реляционную базу данных и новую NoSQL базу. При таком подходе в базу данных для чтения ведется запись именно в том формате, который необходим пользователю, поэтому чтение выполняется быстро

Поскольку архитектура CQRS разделяет операции чтения и записи, использование одной базы данных для операций чтения, а второй для операций записи выглядит логичным. Сторона записи хранит данные в реляционной форме без дублирования данных – процесс, известный как *нормализация*, – а сторона чтения хранит данные в форме, соответствующей пользовательскому интерфейсу.

В приложении Book App сторона чтения будет содержать данные, уже преобразованные в соответствии с потребностями отображения книги; эти предварительно созданные сущности называются *проекциями*. Они созданы с использованием того же кода, что и метод `MapBookToDto` из раздела 2.6. Мы предварительно создаем нужные нам представления и записываем их в базу данных для чтения.

Такой дизайн приложения обеспечивает хороший прирост производительности при чтении, но снижает производительность при записи, что делает архитектуру CQRS с двумя базами данных подходящим вариантом в случае, если у бизнес-приложения больше операций чтения, чем записи. У многих бизнес-приложений операций чтения больше, чем записей (хороший пример – приложения для онлайн-торговли), поэтому эта архитектура хорошо подходит для нашего приложения Book App.

16.4 Проектирование приложения с архитектурой CQRS с двумя базами данных

Фундаментальная проблема при создании любой системы CQRS – убедиться, что любые изменения синхронизированы со связанной проекцией в базе данных CQRS на стороне чтения. Если вы ошибетесь, то пользователь увидит неверные данные. Это та же проблема, что и проблема инвалидации кеша, над которой я так упорно трудился, используя подход с кешированным SQL, описанный в разделе 15.5. Хитрость состоит в том, чтобы перехватывать каждое изменение сущности `Book` и связанных с ней сущностей и гарантировать обновление базы данных CQRS, выполняющей операции чтения.

В первом издании этой книги я обнаруживал изменения в сущности `Book` и связанных сущностях путем просмотра свойства отслеживаемых сущностей `State` при вызове методов `SaveChanges` и `SaveChangesAsync`. Эти свойства `State` и сущности анализировались, чтобы определить, нужно ли добавлять, обновлять или удалять проекцию в NoSQL базе данных. Это допустимый подход (я показываю пример в разделе 12.5), но использование свойства `State` для нескольких сущностей может быть довольно сложным.

Еще один подход заключается в использовании событий интеграции (раздел 12.1.2), инициируемых методами доступа предметно-ориентированного проектирования (DDD) (см. раздел 13.4.2). Вот некоторые преимущества этого подхода:

- *больше надежности* – использование событий интеграции гарантирует, что реляционная база данных обновляется только после того, как Cosmos DB успешно обновила свою базу данных. Применение обновлений обеих баз данных в транзакции снижает вероятность того, что синхронизация реляционной базы данных и Cosmos DB будет нарушена (в первом издании этой книги такое могло произойти, если бы обновление RavenDb окончилось неудачно);
- *больше очевидности* – вы инициируете события интеграции внутри DDD-методов, изменяющих данные. Каждое событие сообщает обработчику событий, является ли оно добавлением, обновлением или удалением (в данном случае – мягким удалением) сущности `Book`. Тогда написать обработчик событий для добавления, обновления или удаления проекции `Book` в Cosmos DB не составит труда;
- *больше простоты* – как уже было сказано, отправка событий интеграции намного проще, чем добавление изменений через свойство `State` отслеживаемых сущностей (см. раздел 12.5, где приводится описание этого подхода).

На рис. 16.2 показано, что происходит, когда администратор добавляет новую сущность `Book`, и как она добавляется в базу данных Cosmos DB и становится доступной пользователям.

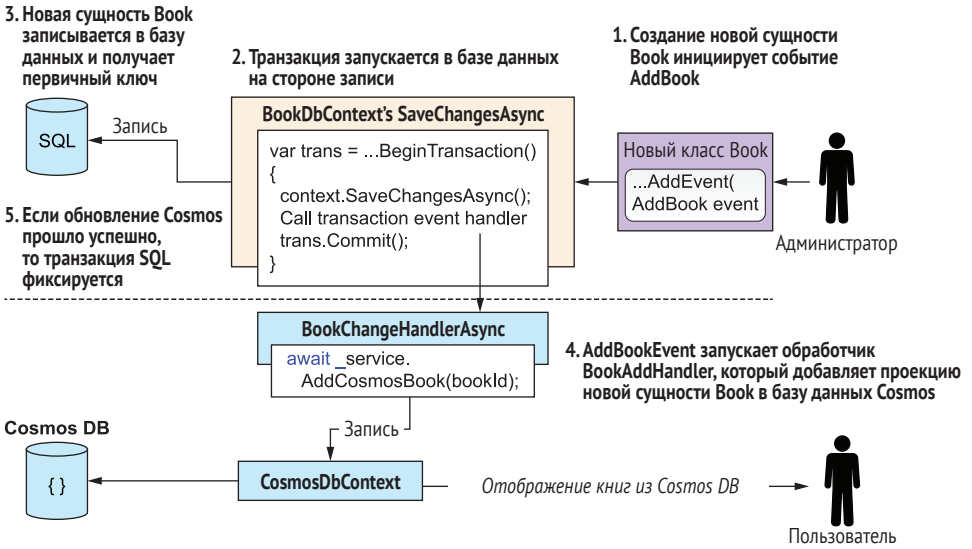


Рис. 16.2 Добавление новой сущности Book. Статический фабричный метод Book добавляет событие интеграции Add Book; оно перехватывается классом BookDbContext, который обрабатывает запросы к реляционной базе данных. Методы SaveChanges и SaveChangesAsync были переопределены библиотекой EfCore.GenericEventRunner. Поскольку это событие интеграции, библиотека запускает транзакцию и записывает новую сущность Book, которая получает первичный ключ. Затем событие интеграции Add Book вызывает обработчик событий BookChange, который создает проекцию новой сущности Book и добавляет ее в базу данных Cosmos DB. Если запись в базу данных Cosmos DB прошла успешно, транзакция фиксируется, и расхождений в базах нет. Если Cosmos DB дает сбой, то транзакция SQL откатывается, и администратор получает предупреждение о том, что добавить новую книгу не удалось

Для реализации системы CQRS, показанной на рис. 16.2, нужно выполнить следующие шаги:

1. Создайте событие, которое будет вызываться при изменении SQL-сущности Book.
2. Добавьте события в сущность Book для отправки событий интеграции Add, Update или Delete.
3. Используйте библиотеку EfCore.GenericEventRunner, чтобы переопределить класс BookDbContext.
4. Создайте классы сущностей для Cosmos и DbContext.
5. Создайте обработчики событий Cosmos Add, Update и Delete.

16.4.1 Создание события, вызываемого при изменении сущности Book

В этом проекте нам нужно обновить базу данных Cosmos DB при обнаружении события интеграции Add, Update или Delete. Но возможно, что когда вы добавляете сущность Book, которая создает событие

Add, вы также можете инициировать событие Update (это происходит при заполнении базы данных). Кроме того, некоторые сложные обновления, такие как изменение нескольких частей сущности, могут инициировать несколько событий Update. Несколько событий как минимум неэффективны, поскольку в этом случае база данных Cosmos DB должна быть обновлена несколько раз и в некоторых случаях код усложняется. Проблема состоит в том, что обработчик событий не знает о других событиях, поэтому нельзя определить, что обновление не требуется. Как свести несколько событий к одному?

Для решения подобных проблем библиотека GenericEventRunner предоставляет атрибут RemoveDuplicateEvents, чтобы удалить повторяющиеся события, которые относятся к одному типу и связаны с одним и тем же экземпляром класса (определяется методом ReferenceEquals). В следующем листинге показано событие BookChangedEvent с добавленным атрибутом RemoveDuplicateEvents.

Листинг 16.1 Событие BookChangedEvent, отправляющее изменения Add, Update и Delete

Этот атрибут заставляет библиотеку GenericEventRunner удалять повторяющиеся события для одного и того же экземпляра Book

Три типа изменений, которые необходимо отобразить в базу данных Cosmos DB

```
public enum BookChangeTypes { Added, Updated, Deleted }

[RemoveDuplicateEvents]
public class BookChangedEvent : IEntityEvent
{
    public BookChangedEvent(BookChangeTypes bookChangeType)
    {
        BookChangeType = bookChangeType;
    }

    public BookChangeTypes BookChangeType { get; }
}
```

При создании события нужно сообщить, через какой тип изменения прошла сущность Book

Содержит тип изменения, которое будет использовать обработчик событий

Используется обработчиком событий для определения необходимости добавления, обновления или удаления CosmosBook

Этот листинг не только более эффективен. Он также упрощает код, обновляющий Cosmos DB, поскольку добавление с последующим обновлением вызовет проблемы с обновлением сущности с тем же ключом, которая уже отслеживается. Эту проблему можно решить с помощью кода Cosmos Add/Update, но удалить повторяющиеся события проще, тем более что эта функция встроена в библиотеку GenericEventRunner.

16.4.2 Добавление событий в метод сущности Book

Поскольку мы используем классы сущностей в DDD-стиле, достаточно легко определить все места, где сущность Book создается или обновляется. Вы просто добавляете событие Added в статический фабричный

метод Book и множество событий Update в любые DDD-методы доступа. В следующем листинге показано событие Update, добавляемое с помощью метода AddEvent (см. раздел 12.4.2), если обновление не было отклонено из-за ошибки пользовательского ввода.

Листинг 16.2 Добавление события BookUpdate к методу AddPromotion сущности Book

```
public IStatusGeneric AddPromotion(
    decimal actualPrice, string promotionalText)
{
    var status = new StatusGenericHandler();
    if (string.IsNullOrWhiteSpace(promotionalText))
    {
        return status.AddError(
            "You must provide text to go with the promotion.",
            nameof(PromotionalText));
    }

    ActualPrice = actualPrice;
    PromotionalText = promotionalText;

    if (status.IsValid)
        AddEvent(new BookChangedEvent(
            BookChangeTypes.Updated),
            EventToSend.DuringSave);
    return status;
}
```

Вы запускаете обновления только в том случае, если изменение было допустимым

Добавляет событие BookChangedEvent с параметром Update как событие (интеграции) During

Для события Delete используется мягкое удаление, поэтому изменение фиксируется в свойстве SoftDeleted с помощью метода доступа. Вот какие есть варианты:

- если значение SoftDeleted не изменилось, то событие не отправляется;
- если значение SoftDeleted меняется на true, отправляется событие Deleted;
- если значение SoftDeleted меняется на false, отправляется событие Added.

Этот пример показан в следующем листинге.

Листинг 16.3 Изменение SoftDeleted, инициирующее событие AddBook или DeleteBook

```
public void AlterSoftDelete(bool softDeleted)
{
    if (SoftDeleted != softDeleted)
    {
        var eventType = softDeleted
            ? BookChangeTypes.Deleted
            : BookChangeTypes.Added;
```

Вы запускаете обновления только в том случае, если было изменение свойства SoftDeleted

Тип отправляемого события зависит от нового значения SoftDelete

```
            AddEvent(new BookChangedEvent(eventType)
                , EventToSend.DuringSave);
        }
        SoftDeleted = softDeleted;
    }
```

Добавляет событие BookChangedEvent как событие (интеграции) During

16.4.3 Использование библиотеки EfCore.GenericEventRunner для переопределения BookDbContext

В разделе 15.5.1 мы использовали подход к настройке производительности «Кешированный SQL». Подход SQL (+ *кеш*) использует события предметной области, а подход с CQRS использует события интеграции. «Кешированный SQL» и CQRS могут сосуществовать, причем ничего не зная друг о друге, – еще один пример применения принципа разделения ответственностей.

16.4.4 Создание классов сущностей Cosmos и DbContext

Проекция SQL-сущности `Book` должна содержать обычные свойства, например `Title` и `AcualPrice`, а также значения, на вычисление которых требуется много времени, например количество `Review`, связанных с `Book`. Идея состоит в том, чтобы создать готовую к отображению версию `Book`, известную как *проекция*, чтобы ее можно было быстро получать. В следующих двух листингах показаны классы `CosmosBook` и `CosmosTag`, которые используются для хранения проекции `Book`.

Листинг 16.4 Класс `CosmosBook`, содержащий проекцию SQL-сущности `Book`

```
public class CosmosBook
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
    public bool EstimatedDate { get; set; }
    public int YearPublished { get; set; }
    public decimal OrgPrice { get; set; }
    public decimal ActualPrice { get; set; }
    public string PromotionalText { get; set; }
    public string ManningBookUrl { get; set; }

    public string AuthorsOrdered { get; set; }
    public int ReviewsCount { get; set; }
    public double? ReviewsAverageVotes { get; set; }

    public List<CosmosTag> Tags { get; set; }
    public string TagsString { get; set; }
}
```

Мы используем BookId для связи этой сущности с сущностью в реляционной базе данных

Обычные свойства, необходимые для отображения Book

Предварительно вычисленные значения, используемые для отображения и фильтрации

Эта строка используется позже, чтобы преодолеть ограничение текущего поставщика Cosmos DB

Чтобы осуществлять фильтрацию по TagId, мы предоставляем список CosmosTag, которые являются собственными типами

Листинг 16.5 Класс CosmosTag, содержащий TagId из SQL-версии сущности Book

```
public class CosmosTag
{
    public string TagId { get; set; }
}
```

Как видно из этого листинга, класс `CosmosTag` содержит одно свойство: `TagId`. Он имитирует класс `Tag`, используемый в реляционной базе данных, но будет добавлен как собственный тип (см. раздел 8.9.1). Коллекция `Tags` в `CosmosBook` содержит каждую строку `Tag` для `Book`, позволяя фильтровать книги по тегу, например книги о базах данных по тегу `"Databases"`. Класс `CosmosTag` зарегистрирован как собственный тип, поэтому встроен в данные, отправляемые в Cosmos DB (см. листинг 16.10).

Фактически распространенный способ сохранения данных в Cosmos DB – это хранение коллекции других классов внутри основного класса (в Cosmos DB этот подход называется *вложенностью*), а именно это и делают собственные типы EF Core. Рассмотрите возможность использования классов вложенных собственных типов при создании данных, которые должны храниться в Cosmos DB.

DbContext Cosmos для EF Core небольшой и простой, как показано в листинге 16.6, потому что многие команды конфигурации EF Core не работают с Cosmos DB. Вы не можете настроить тип сохраняемых данных, поскольку каждое свойство преобразуется в ключ/значение JSON, а другие параметры, такие как индексирование, обрабатываются самой Cosmos.

Листинг 16.6 Класс DbContext, необходимый для доступа к базе данных Cosmos DB

```
public class CosmosDbContext : DbContext
{
    public CosmosDbContext(
        DbContextOptions<CosmosDbContext> options)
        : base(options)
    { }

    public DbSet<CosmosBook> Books { get; set; }
    protected override void OnModelCreating(
        ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<CosmosBook>()
            .HasKey(x => x.BookId);

        modelBuilder.Entity<CosmosBook>()
            .OwnsMany(p => p.Tags);
    }
}
```

DbContext Cosmos DB имеет ту же структуру, что и любой другой DbContext

Необходимо, чтобы вы могли читать и записывать CosmosBooks

BookId не соответствует правилам «По соглашению», поэтому нужно настроить его вручную

Коллекция CosmosTag принадлежит CosmosBook

ПРИМЕЧАНИЕ Полный список команд Fluent API для Cosmos DB см. в документации EF Core по провайдеру базы данных Cosmos на странице http://mng.bz/8WyK.

16.4.5 Создание обработчиков событий Cosmos

Событие интеграции `BookChangedEvent` входит в `BookDbContext`, и нам нужен соответствующий обработчик событий. Свойство `BookChangeType` указывает, что это за событие: `Add`, `Update` или `Delete`, поэтому использует оператор C# `switch` для вызова правильного кода. Поскольку при добавлении, обновлении и удалении записей в базе данных Cosmos используется аналогичный код, вы создаете сервис, который содержит три метода, по одному для каждого типа обновления. Размещение всего кода обновления в сервисе делает обработчик событий проще, как показано в следующем листинге.

Листинг 16.7 Пример обработчика событий Cosmos, который обрабатывает событие Add

Определяет класс как событие (интеграции) During для события BookChanged

Этот сервис предоставляет код для добавления, обновления и удаления CosmosBook

```
public class BookChangeHandlerAsync
    : IDuringSaveEventHandlerAsync<BookChangedEvent>
{
    private readonly IBookToCosmosBookService _service;

    public BookChangeHandlerAsync(
        IBookToCosmosBookService service)
    {
        _service = service;
    }

    public async Task<IStatusGeneric> HandleAsync(
        object callingEntity, BookChangedEvent domainEvent,
        Guid uniqueKey)
    {
        var bookId = ((Book)callingEntity).BookId;
        switch (domainEvent.BookChangeType)
        {
            case BookChangeTypes.Added:
                await _service.AddCosmosBookAsync(bookId);
                break;
            case BookChangeTypes.Updated:
                await _service.UpdateCosmosBookAsync(bookId);
                break;
            case BookChangeTypes.Deleted:
                await _service.DeleteCosmosBookAsync(bookId);
                break;
            default:
                throw new ArgumentOutOfRangeException();
```

Обработчик событий использует асинхронный режим, поскольку его использует Cosmos DB

Извлекает BookId из вызывающей сущности, т. е. Book

BookChangeType может быть Added, Updated или Deleted

Вызывает метод сервиса для добавления с параметром BookId сущности Book

Вызывает метод сервиса для обновления с параметром BookId сущности Book

Вызывает метод сервиса для удаления с параметром BookId сущности Book

```
        }
        return null;
    }
}
```

Возвращая null, мы сообщаем GenericEventRunner, что этот метод всегда успешен

Помните, что в случае сбоя обновления базы данных Cosmos обновление SQL, которое было выполнено в транзакции, откатывается, поэтому в базах данных не появляется расхождений. Но нам нужно свести к минимуму вероятность исключения, если сервис может сам решить проблему, сделав дополнительные проверки, чтобы перехватывать состояния, которые можно исправить.

В следующем листинге показан метод `MapBookToCosmosBookAsync`, который обрабатывает обновление `Book`. Маловероятно, но за то время, пока обработчик события `Update` вызывал SQL-код, сущность `Book`, возможно, была (мягко) удалена. Поэтому если метод `MapBookToCosmosBookAsync` возвращает значение `null`, он предполагает, что `Book` была удалена, и удаляет все существующие `CosmosBook` с этим `BookId`. Обратите внимание на использование метода EF Core `Update` в коде.

Листинг 16.8 Создание проекции SQL-сущности Book и добавление ее в базу данных Cosmos

Этот метод вызывается обработчиком события BookUpdated с параметром BookId сущности Book

```
public async Task UpdateCosmosBookAsync(int bookId)
{
    if (CosmosNotConfigured)
        return;

    var cosmosBook = await MapBookToCosmosBookAsync(bookId);

    if (cosmosBook != null)
    {
        _cosmosContext.Update(cosmosBook);
        await CosmosSaveChangesWithChecksAsync(
            WhatDoing.Updating, bookId);
    }
    else
    {
        await DeleteCosmosBookAsync(bookId);
    }
}
```

Приложение Book App можно запускать без доступа к Cosmos DB, и в этом случае метод тотчас же завершит работу

В этом методе используется метод Select, аналогичный тому, который описан в главе 2 для класса сущности CosmosBook

Если CosmosBook успешно заполнена, выполняется код обновления Cosmos

Обновляет CosmosBook через cosmosContext, а затем вызывает метод для сохранения его в базе данных

Если SQL-сущность Book не была найдена, мы гарантируем, что проекция будет удалена из базы данных Cosmos

ОТЛИЧИЕ В COSMOS DB База данных Cosmos DB всегда обновляет всю запись для данного ключа за один раз, в отличие от реляционной базы данных, которая может изменять отдельные столбцы в строке. Метод EF Core `Update` более эффективный, поскольку не выполняет операцию чтения из базы данных Cosmos.

Метод `CosmosSaveChangesWithChecksAsync` также предназначен для перехвата и исправления любых некорректных состояний, которые он может обнаружить. Например, при обновлении, если вы не нашли сущность `CosmosBook`, которую нужно обновить, будет выполнено создание новой сущности `CosmosBook`. Такие ситуации – редкость, но могут происходить из-за одновременных обновлений одной и той же сущности `CosmosBook`.

В листинге 16.9 показана часть метода `CosmosSaveChangesWithChecksAsync`, которая обнаруживает ошибки, возможно, вызванные проблемами параллельного доступа, обеспечивая актуальность базы данных Cosmos. Часть кода с `catch` в листинге обрабатывает следующие ситуации:

- `CosmosException`:
  - `Update`, в котором обновляемая сущность была удалена, `Update` преобразуется в `Add`;
  - `Delete`, в котором удаляемая сущность уже была удалена (работа выполнена);
  - если исправлений не было, повторно выбрасывается исключение;
- `DbUpdateException`:
  - `Add`, добавляемая сущность уже существует, `Add` преобразуется в `Update`.

Этот код показывает еще одно полезное отличие при использовании провайдера Cosmos DB.

Листинг 16.9 Часть обработки исключений SaveChanges с Cosmos DB

Вызывает метод SaveChanges и обрабатывает определенные ситуации

```
private async Task CosmosSaveChangesWithChecksAsync
    (WhatDoing whatDoing, int bookId)
{
    try
    {
        await _cosmosContext.SaveChangesAsync();
    }
    catch (CosmosException e)
    {
        if (e.StatusCode == HttpStatusCode.NotFound
            && whatDoing == WhatDoing.Updating)
        {
            var updateVersion = _cosmosContext
                .Find<CosmosBook>(bookId);
            _cosmosContext.Entry(updateVersion)
                .State = EntityState.Detached;

            await AddCosmosBookAsync(bookId);
        }
```

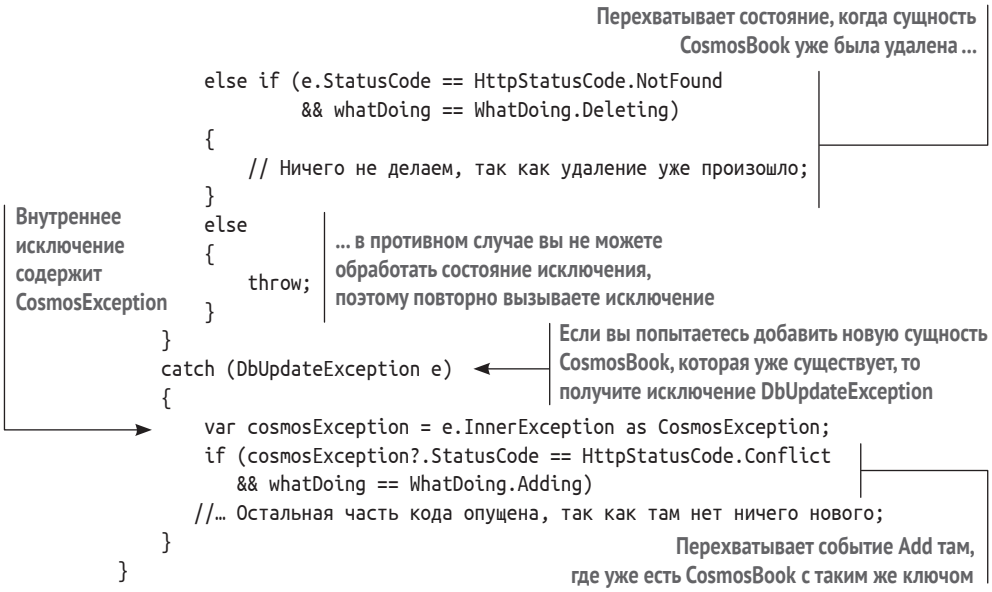
Параметр whatDoing сообщает методу, идет ли речь о добавлении, обновлении или удалении

Перехватывает попытку обновить сущность CosmosBook, которой там не было

Перехватывает любые исключения CosmosException

Необходимо удалить обновление; в противном случае EF Core выдаст исключение

Преобразует обновление в добавление



ОТЛИЧИЕ В COSMOS DB Я обнаружил, что `CosmosException` помогает диагностировать проблемы с базой данных Cosmos. `CosmosException` содержит свойство `StatusCode`, использующее коды состояния HTTP, например `NotFound` и `Conflict`, чтобы описать, что пошло не так.

16.5 Структура и данные учетной записи Cosmos DB

Прежде чем перейти к запросу класса `CosmosBook`, стоит посмотреть, как организована Cosmos DB и как выглядят данные при записи EF Core в базу данных. В этих разделах объясняется, как использовать учетную запись для доступа к базе данных Cosmos в своем приложении и просмотреть хранящиеся в ней данные в формате JSON.

ПРИМЕЧАНИЕ Провайдер EF Core использует Cosmos SQL API, который представляет собой традиционное NoSQL хранилище документов с использованием JSON. Но в Cosmos DB есть несколько способов обработки данных, например хранилище столбцов; пары «ключ/значение» и графы; и несколько API, таких как MongoDB, Cassandra, Azure Table и Gremlin (графы).

16.5.1 Структура Cosmos DB с точки зрения EF Core

В этом разделе представлен краткий обзор различных частей структуры Cosmos DB. Это не подробное объяснение (более полное есть

в документации по Azure), но оно предоставляет термины, необходимые для использования Cosmos DB с EF Core.

Azure предоставляет учетную запись Azure Cosmos DB, которая похожа на сервер базы данных, поскольку у вас может быть несколько баз данных в одной учетной записи Azure Cosmos DB. Доступ к этой учетной записи можно получить через строку подключения, состоящую из двух частей: URI для доступа к учетной записи Cosmos DB и ключа учетной записи. Эта комбинация позволяет получить доступ к учетной записи Azure Cosmos DB.

ПРИМЕЧАНИЕ Эмулятор Azure Cosmos DB предоставляет локальную (и бесплатную) версию учетной записи Cosmos DB. Кроме того, он также содержит функциональность, позволяющую читать и управлять базами данных, которые хранятся локально. Я расскажу об эмуляторе Azure Cosmos DB в разделе 17.8.

Учетная запись Cosmos DB может содержать много баз данных Cosmos DB; у каждой может быть множество контейнеров Cosmos DB; а контейнеры – это место, где хранятся данные. На рис. 16.3 показано, как код EF Core отображается в структуру Cosmos DB.

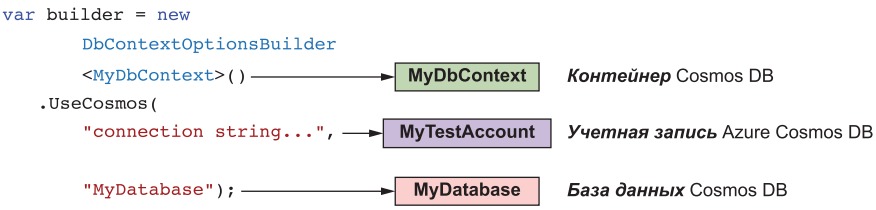


Рис. 16.3 Отображение настройки EF Core поставщика базы данных Cosmos DB в три уровня в системе Cosmos DB. Учетная запись Azure Cosmos DB может иметь много баз данных Cosmos DB, но на этом рисунке показана только одна. Имя базы данных определяется в методе `UseCosmos`. У базы данных Cosmos DB может быть много контейнеров, но при использовании EF Core она позволяет применять только один контейнер для каждого DbContext приложения EF Core. По умолчанию контейнеру присваивается имя класса DbContext приложения

Настроив класс `DbContextOptionsBuilder<T>` (или зарегистрировав DbContext Cosmos через метод `AddDbContext`), можно получить экземпляр DbContext приложения и получить доступ к базе данных Cosmos DB.

16.5.2 Как CosmosClass хранится в Cosmos DB

Если у вас есть правильно настроенный DbContext для базы данных Cosmos DB, вы можете выполнять чтение и запись в базу данных – строго говоря, контейнер Cosmos DB, но применительно к EF Core я буду использовать слово *база данных*. Для обычных операций чте-

ния или записи не нужно знать, как данные хранятся в базе данных Cosmos DB, но иногда эта информация бывает полезной, поскольку дает представление о том, что там хранится.

В следующем листинге показано, как хранятся данные после записи сущности `CosmosBook` в базу данных. Как видите, дополнительные свойства в конце не входят в класс `CosmosBook`, но они критически важны для работы Cosmos DB.

Листинг 16.10 Данные `CosmosBook`, хранящиеся в формате JSON в Cosmos DB

```
{
  "BookId": 214,
  "ActualPrice": 59.99,
  "AuthorsOrdered": "Jon P Smith",
  "EstimatedDate": true,
  "ManningBookUrl": "
  "OrgPrice": 59.99,
  "PromotionalText": null,
  "PublishedOn": "2021-05-15T05:00:00+01:00",
  "ReviewsAverageVotes": 5,
  "ReviewsCount": 1,
  "Title": "Entity Framework Core in Action, Second Edition",

  "Tags": [
    {
      "TagId": "Databases"
    },
    {
      "TagId": "Microsoft & .NET"
    }
  ],

  "YearPublished": 2021,
  "TagsString": "| Databases | Microsoft & .NET |",

  "Discriminator": "CosmosBook",

  "id": "CosmosBook|214",

  "_rid": "QmRlAMizcQmwAg…",
  "_self": "dbs/QmRlAA==/colls/QmRlAMizcQk=…",
  "_etag": "\"1e01b788-0000-1100-0000-5facfa2f0000\"",
  "_ts": 1605171759,
  "_attachments": "attachments/"
}
```

Стандартные свойства из класса CosmosBook

Содержит коллекцию Tags, которые являются собственными типами

Эти два свойства добавлены для преодоления ограничений в поставщике EF Core 5 для Cosmos

Id – это первичный ключ базы данных, и он должен быть уникальным. Устанавливается EF Core с использованием назначенного первичного ключа и дискриминатора

Свойства для Cosmos; см. следующие примечания

EF Core добавляет дискриминатор, чтобы отличать этот класс от других классов, сохраненных в том же контейнере Cosmos

Первый набор ключей и значений JSON идет из свойств и связей в классе `CosmosBook`, включая коллекцию Tags:

- пара «ключ/значение» `id` – это уникальный ключ, используемый для определения этих данных. EF Core заполняет уникальный ключ

значением – по умолчанию это комбинация значения `Discriminator` и значения из свойства (свойств), которые, как вы сообщили EF Core, являются первичным ключом этого класса сущности;
- пару «ключ/значение» `_etag` можно использовать с методом Fluent API `UseETagConcurrency` для предоставления маркера параллелизма, применяемого к любому изменению данных;
- пара «ключ/значение» `_ts` содержит время последнего добавления/обновления в формате Unix и полезна для определения даты последнего изменения записи. Значение `_ts` можно преобразовать в формат `DateTime` в C# с помощью класса `UnixDateTimeConverter`;
- пары «ключ/значение» `_rid` и `_self` – это уникальные идентификаторы, используемые внутри для навигации и ресурсов;
- пара «ключ/значение» `_attachments` устарела и используется только для старых систем.

16.6 Отображение книг через Cosmos DB

Создав систему, которая копирует изменения в классе сущности SQL `Book` в базу данных Cosmos, теперь можно реализовать функции показа книг для исходного приложения Book App, получая данные из базы данных Cosmos DB. Реализация этих возможностей раскрывает несколько интересных отличий Cosmos DB от реляционных баз данных.

В конце концов, у меня все получилось, но интересно понять, чего можно достичь, используя базу данных Cosmos DB. Я создал отображение книг с прямыми командами Cosmos DB, используя ее .NET SDK (комплект для разработки программного обеспечения), который я называю Cosmos (Direct). Код Cosmos (Direct) позволил мне разграничить ограничения провайдера базы данных EF Core 5 для Cosmos и различия в способах, которыми Cosmos DB изначально выполняет запрос к базе данных.

> **ОГРАНИЧЕНИЕ EF CORE 5** SQL-команды EF Core, например `FromSqlRaw` и `FromSqlInterpolated`, не работают. Но можно получить экземпляр `CosmosClient` через `var cosmosClient = context.Database.GetCosmosClient()`. Этот способ позволяет использовать команды пакета .NET SDK для Cosmos DB.

Далее описываются отличия от реляционных баз данных и ограничения EF Core 5, которые я обнаружил при реализации архитектуры CQRS с двумя базами данных:

- отличия Cosmos DB от реляционных баз данных;
- основное различие Cosmos DB и EF Core: миграция базы данных Cosmos;
- ограничения провайдера базы данных EF Core 5 для Cosmos DB.

ПРИМЕЧАНИЕ Если вы хотите попробовать запустить приложение Book App с Cosmos DB, скачайте связанный репозиторий GitHub (http://mng.bz/XdlG), запустите проект BookApp.UI и поищите ссылку «Chapter 16 Setup» на домашней странице для получения дополнительной информации.

16.6.1 Отличия Cosmos DB от реляционных баз данных

В этом разделе рассматриваются различия между базой данных Cosmos DB и реляционной базой данных (SQL Server). Эта информация будет полезна разработчикам, которые прежде еще не работали с NoSQL базами данных, а точнее с базой данных Cosmos DB. Вот краткая сводка этих отличий:

- Cosmos DB предоставляет только асинхронные методы;
- отсутствие первичных ключей, создаваемых базой данных;
- сложные запросы могут нуждаться в разбивке;
- метод `Skip` медленный и затратный;
- по умолчанию проиндексированы все свойства.

COSMOS DB ПРЕДОСТАВЛЯЕТ ТОЛЬКО АСИНХРОННЫЕ МЕТОДЫ

Поскольку Cosmos DB использует протокол HTTP для доступа к базам данных, все методы в .NET SDK применяют async/await, а синхронные версии отсутствуют. EF Core предоставляет доступ к Cosmos DB через синхронные методы EF Core, такие как `ToList` и `SaveChanges`, но в настоящее время эти методы используют метод `Task.Wait`, у которого могут быть проблемы с взаимоблокировками.

Я настоятельно рекомендую использовать только асинхронные методы EF Core при работе с провайдером базы данных Cosmos. Помимо более надежного приложения, вы получите лучшую масштабируемость в многопользовательских средах, таких как ASP.NET Core.

ОТСУТСТВИЕ ПЕРВИЧНЫХ КЛЮЧЕЙ, СОЗДАВАЕМЫХ БАЗОЙ ДАННЫХ

При использовании реляционных баз данных мы привыкли, что база предоставляет уникальное значение для своего первичного ключа при добавлении новой строки в таблицу. Но в Cosmos и многих других NoSQL базах данных по умолчанию ключ для элемента (элемент – это обозначение, используемое Cosmos для каждой записи в формате JSON) должен создаваться программой перед добавлением элемента.

ПРИМЕЧАНИЕ В Cosmos DB есть способ создать уникальный ключ, но он будет храниться в паре «ключ/значение» `id`.

Ключ элемента должен быть уникальным, иначе Cosmos DB отклонит (используя код состояния HTTP `Conflict`) создание нового эле-

мента, если его ключ уже использовался. Кроме того, после того как вы добавили элемент с ключом, изменить ключ нельзя.

Один из простых вариантов для ключа Cosmos DB – это тип C# `Guid`, который спроектирован, чтобы быть уникальным. Кроме того, EF Core упрощает использование этого типа в качестве ключа, поскольку у него есть встроенный генератор значений (см. раздел 10.3.3), который предоставит новое значение `Guid`, если первичный ключ – это `Guid`, а его значение – `default`. Составные ключи можно настроить с помощью EF Core, который объединит их значения в строку, необходимую Cosmos DB для пары «ключ/значение» `id`. При использовании Cosmos в приложении Book App я воспользовался `int` в качестве ключа для сущности `CosmosBook`, но значение `int` получено из первичного ключа, созданного реляционной базой данных на стороне записи.

> **ПРИМЕЧАНИЕ** В Cosmos DB говорится о ключе секции, а также о логических и физических секциях. Я не затрагиваю здесь эти вопросы, поскольку это обширные темы и я не уверен, что достаточно хорошо разбираюсь в них. В EF Core 5 по умолчанию нет ключа секции, но эту настройку можно изменить.

СЛОЖНЫЕ ЗАПРОСЫ МОГУТ НУЖДАТЬСЯ В РАЗБИВКЕ

В параметре «Фильтр по годам» при отображении книг `FilterDropdownService` находит все года публикации книг. Эта задача требует ряда шагов:

1. Отфильтровать все книги, которые еще не были опубликованы.
2. Извлечь `Year` из свойства `DateTime PublishedOn` сущности `Book`.
3. Применить LINQ-команду `Distinct`, чтобы получить уникальные даты для всех опубликованных книг.
4. Упорядочить даты.

Такой сложный запрос работает в реляционной базе данных, но Cosmos DB не справляется с ним. На рис. 16.4 показаны оба варианта запроса.

Когда я запустил код, который использовал в SQL Server (см. правую часть рис.16.4), то получил исключение Cosmos DB со ссылкой на проблему EF Core #16156, где говорится, что в Cosmos DB имеются некоторые ограничения на запросы. Cosmos не поддерживает функциональность запросов с большой вложенностью, который реляционные базы данных приобрели за десятилетия развития, поэтому вам, возможно, придется изменить некоторые сложные запросы при работе с этой базой данных. Вот что сделал я, чтобы запрос, получающий значения для фильтрации по раскрывающемуся списку, работал в Cosmos DB:

- я добавил новое свойство `YearPublished`, в котором год был указан как целое число. (Я попытался использовать пользовательскую функцию Cosmos DB для извлечения года, но это бы не сработало с командой `Distinct`.) Это свойство заполняется во время

проекции SQL-сущности `Book` из значения `Year` свойства `DateTime PublishedOn`;

- я выполнил запрос с командой `Distinct`, используя значение `YearPublished` в Cosmos, а затем упорядочил возвращаемые годы в приложении.

Здесь показаны две версии FilterDropdownService для поиска всех дат публикаций книг

Пример с Cosmos DB

```
var nextYear
var al          await_db
    .Se
    .Di

var re
    .Where(x  > x
    .OrderByDescending(x
    .Se           new DropdownTuple
    {
        Value
        Text
    }).ToList();
```

Пример с SQL Server

```
var nextYear
var re
    .Where(x
    .Se
    .Di
    .Where(x  > x
    .OrderByDescending(x
    .Se           new DropdownTuple
    {
        Value
        Text
    }).ToList();
```

Рис. 16.4 Две версии сервиса `FilterDropdownService`, который находит все годы публикации книг. Пример с Cosmos DB упрощает запрос, выполняемый в Cosmos DB, а вторая часть выполняется в приложении. Этот пример показывает, что у Cosmos DB нет широкого набора функций запросов, которые есть у реляционных баз данных

Два моих изменения в коде заставляют запрос Cosmos работать, но он работает медленно (раздел 16.7.2). Вывод заключается в том, что не следует применять запросы из нескольких частей к базе данных Cosmos DB, независимо от того, используете вы EF Core или нет. Сильной стороной базы данных Cosmos DB является ее масштабируемость и доступность, а не способность обрабатывать сложные запросы.

МЕТОД SKIP – МЕДЛЕННЫЙ И ЗАТРАТНЫЙ

В приложении Book App я использовал разбиение на страницы, чтобы пользователь мог перемещаться по страницам, где показаны книги. Этот тип запроса применяет LINQ-методы `Skip` и `Take`, чтобы обеспечить разбиение на страницы. Запрос `context.Books.Skip(100).Take(10)`, например, вернет с 101-ю по 111-ю книги в последовательности. Cosmos DB тоже может это делать, но часть, где используется метод `Skip`, работает медленнее, по мере того как значение skip растет (еще одно отличие от реляционных баз данных). Кроме того, это затратно.

ЕДИНИЦЫ ЗАПРОСА COSMOS База данных Azure Cosmos DB использует *единицы запроса* для управления подготовленной пропускной способностью контейнера. Есть разные способы для подготовки контейнера Cosmos DB: фиксированная подготовка (фиксированная цена), бессерверный (оплата по мере ис-

пользования) и автоматическое масштабирование (масштабирование для использования). Однако в итоге вы будете платить за каждый доступ к службе Cosmos DB.

Кажется, что если пропустить 100 элементов, Cosmos все равно их прочитает. Но хотя Cosmos не отправляет пропущенные элементы в приложение, единицы запроса – это время и деньги. В приложении Book App видно, что производительность снижается по мере того, как пользователь идет вниз по списку книг (см. рис. 16.9).

Проблема с производительностью `Skip` зависит от приложения. В приложении Book App я сомневаюсь, что люди прочитают намного больше первых 100 книг. Но этот пример предполагает, что лучше показать 100 книг за раз, чем выводить по 10 книг на странице, поскольку разбиение на страницы – вещь не бесплатная.

ПО УМОЛЧАНИЮ ПРОИНДЕКСИРОВАНЫ ВСЕ СВОЙСТВА

Мы знаем, что добавление индекса к свойству в реляционной базе данных значительно сокращает время, необходимое для фильтрации или сортировки по этому свойству, с (небольшими) затратами на производительность при обновлении проиндексированного свойства. По умолчанию Cosmos DB индексирует все пары «ключ/значение», включая вложенные. (У сущности `CosmosBook`, например, пары «ключ/значения» `Tags.TagId` тоже проиндексированы.) Можно изменить политику индексирования Cosmos DB, но «индексировать все» – неплохая отправная точка.

ПРИМЕЧАНИЕ Функции конфигурации индексирования EF Core, включая уникальный индекс, не работают в Cosmos DB. Но можно определить индексы через настройки контейнера.

Кроме того, нужно помнить, что Cosmos DB сохраняет данные, используя строковый формат JSON, а Cosmos известно только три типа индексов: числа, строки и геоданные. Типы C# `DateTime` и `TimeSpan` хранятся в строковом формате, который можно сортировать или фильтровать как строки, поэтому дата и время сохраняются с более значительными временными частями, например `YYYY-MM-DDTHH:MM:SS`. EF Core выполняет преобразование времени в строку за вас, но если вы используете преобразователи значений EF Core (см. раздел 7.8) или SQL-запросы, то необходимо понимать различные форматы JSON, используемые Cosmos DB.

16.6.2 Основное различие между Cosmos DB и EF Core: миграция базы данных Cosmos

Cosmos DB – это база данных *без схемы*. Это означает, что каждый элемент не обязательно должен иметь одинаковые свойства или вложенные данные в каждом элементе. Каждый элемент представляет собой

объект JSON, и вам решать, какие ключи и значения добавлять в него. Эта база данных отличается от реляционной, где важна схема и которая требует усилий для изменений (см. главу 9).

В какой-то момент вы соберетесь изменить или добавить свойства к классу сущности, отображенному в базу данных Cosmos DB. Однако вы должны быть осторожны; в противном случае можно нарушить работу некоторых существующих запросов Cosmos DB. В этом примере показано, что может пойти не так и как это исправить:

1. У вас есть класс сущности `CosmosBook`, и вы записали данные в базу данных Cosmos DB.
2. Вы решили, что вам нужно дополнительное свойство с именем `NewProperty` типа `int` (но это может быть любой тип, не допускающий значения `null`).
3. Вы читаете старые данные, которые были добавлены до того, как свойство `NewProperty` было добавлено в класс сущности `CosmosBook`.
4. На этом этапе вы получаете исключение, в котором говорится что-то вроде `object must have a value`.

Cosmos DB не возражает против того, чтобы в каждом элементе были разные данные, в отличие от EF Core. EF Core ожидает свойство `NewProperty` типа `int`, а его там нет. Чтобы решить эту проблему, необходимо убедиться, что все новые свойства допускают значение `null`; тогда чтение старых данных вернет значение `null` для новых свойств. Если вы хотите, чтобы новое свойство не допускало значения `null`, начните с версии, допускающей значение `null`, а затем обновите каждый элемент в базе данных значением, отличным от `null` для нового свойства. После этого можно изменить тип нового свойства обратно на тип, не допускающий значения `null`, и поскольку в каждом элементе для этого свойства есть значение, все ваши запросы будут работать.

Еще один момент: нельзя использовать команду `Migrate` для создания новой базы данных Cosmos DB, поскольку EF Core не поддерживает миграции для базы данных Cosmos DB. Вместо этого нужно использовать метод `EnsureCreatedAsync`. Обычно этот метод используется для модульного тестирования, но это рекомендуемый способ создания базы данных (контейнера Cosmos DB) при работе с Cosmos DB.

16.6.3 Ограничения поставщика базы данных EF Core 5 для Cosmos DB

В этом разделе рассматриваются ограничения поставщика базы данных EF Core 5 для Cosmos DB. Это полезная информация, если вы хотите использовать EF Core 5 для доступа к базе данных Cosmos DB; кроме того, это будет полезно, когда в будущих версиях EF Core будут устранены некоторые из этих ограничений, что сделает ненужными обходные пути, которые мне пришлось применять к версии приложения Book App для третьей части. Вот краткое изложение ограничений:

- подсчет количества книг в Cosmos DB идет *медленно*!
- многие функции базы данных не реализованы;
- EF Core 5 не может выполнять подзапросы в базе данных Cosmos DB;
- нет никаких связей или методов `Include`.

ПОДСЧЕТ КОЛИЧЕСТВА КНИГ В COSMOS DB ИДЕТ МЕДЛЕННО

Практически первое, что я заметил, когда добавил версию с Cosmos в приложение Book App, – что подсчет сущностей `CosmosBook`, который я использовал для разбиения по страницам, был очень медленным с EF Core. Я создал мини-версию этого приложения в конце 2019 года, и причин плохой производительности было две:

- агрегаты Cosmos DB (`Count`, `Sum` и т. д.) работали медленно, и им требовалось много единиц запросов для работы;
- EF Core не использовала агрегаты Cosmos DB, поэтому мне нужно было считать каждую сущность `CosmosBook`, чтобы посчитать их (ограничение EF Core 5).

К счастью, первая проблема была исправлена в апреле 2020 г. Агрегаты Cosmos DB работают намного быстрее и потребляют гораздо меньше ресурсов. (Пример: исходный агрегат `Count` занял 12 000 единиц запроса, тогда как новый использовал только 25 единиц.) Но EF Core 5 не стал быстрее, потому что получал все книги в базе данных Cosmos, чтобы их посчитать. Чтобы исправить это, я изменил способ работы показа книг Cosmos EF и перешел к использованию подхода к разбиению по страницам под названием `Next/Previous`. На рис. 16.5 показан этот формат.

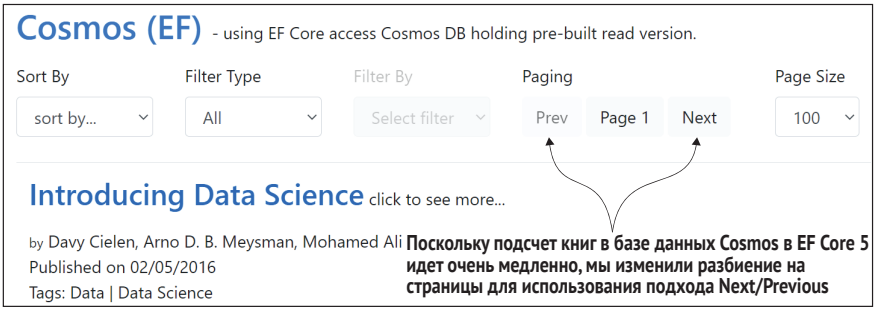


Рис. 16.5 Страница, на которой доступ к Cosmos DB осуществляется через EF Core 5. Чтобы преодолеть медленную скорость подсчета количества книг, я изменил элементы управления, дабы использовать подход `Next/Previous`

Этот переход был тривиальным; основная проблема заключалась в настройке страницы Razor в ASP.NET Core. Многие сайты онлайн-торговли, включая Amazon, используют этот подход, так что это изменение может быть полезным в любом случае.

В версии Cosmos (Direct) я оставил обычное разбиение на страницы с подсчетом всех отфильтрованных книг. Оказывается, прямая

команда Cosmos с `Count` (`SELECT VALUE COUNT(c) FROM c`) выполняется быстро (~ 25 мс для подсчета 500 000 книг) даже по сравнению с SQL-версией (90 мс для подсчета 500 000 книг).

МНОГИЕ ФУНКЦИИ БАЗЫ ДАННЫХ НЕ РЕАЛИЗОВАНЫ

EF Core 5 отображал LINQ в небольшой набор из пяти функций Cosmos, поэтому вам, возможно, придется изменить LINQ, чтобы обойти эти проблемы. Один из фильтров, который я опробовал, дал сбой, потому что EF Core знал, что должен преобразовать метод `DateTime.UtcNow` в дату в формате UTC из сервера базы данных, но эта функция Cosmos DB не была отображена в EF Core 5. Эту проблему было легко исправить: я создал переменную для хранения значения, заданного методом `DateTime.UtcNow`. На рис. 16.6 показаны неудачный (слева) и исправленный (справа) запросы. Различия выделены жирным шрифтом.

НЕУДАЧНЫЙ запрос к Cosmos DB

```
var filterYear = int.Parse(filterValue);
var result = _db.books.Where(x =>
     x.PublishedOn.Year == filterYear &&
     x.PublishedOn <= DateTime.UtcNow
```

ИСПРАВЛЕННЫЙ запрос к Cosmos DB

```
var now = DateTime.UtcNow;
var filterYear = int.Parse(filterValue);
var result = _db.books.Where(x =>
     x.PublishedOn.Year == filterYear &&
     x.PublishedOn <= now
```

Рис. 16.6 Исходный запрос слева не сработал, потому что EF Core 5 знал, что он должен преобразовать метод `DateTime.UtcNow` (слева, жирным шрифтом) в дату в формате UTC, предоставленную сервером базы данных, но эта часть отображения не была выполнена. Решение заключалось в том, чтобы поместить значение из метода `DateTime.UtcNow` в переменную (вверху справа, выделена жирным шрифтом) и добавить ее в запрос

EF CORE 5 НЕ МОЖЕТ ВЫПОЛНЯТЬ ПОДЗАПРОСЫ В БАЗЕ ДАННЫХ COSMOS DB

Приложение Book App позволяет фильтровать книги по тегам, например просматривать только книги с тегом `"Databases"`. Это решение требует наличия вложенного запроса в главном запросе, как SQL-команда для Cosmos DB, показанная в следующем фрагменте кода:

```
SELECT DISTINCT VALUE f.TagId FROM c JOIN f in c.Tags
```

Такое ограничение EF Core 5 не позволяет запрашивать какие-либо вложенные части Cosmos JSON, например собственные типы, которые сохраняются с основным классом сущности. Конечно, эти данные можно получить, читая сущность, но вы не можете фильтровать, сортировать или выбирать вложенные части сами по себе, используя EF Core. В разделе 16.7.2 я покажу способ обойти эту проблему.

ОТСУТСТВИЕ СВЯЗЕЙ ИЛИ МЕТОДОВ INCLUDE

Провайдер баз данных EF Core 5 для Cosmos DB не поддерживает связи между классами сущностей (кроме собственных типов, встроен-

ных в основной класс сущности). Хотя такое отсутствие поддержки кажется большим недостатком, когда дело доходит до сущностей Cosmos, собственные типы – лучший вариант, так что, возможно, это и не так важно.

Подход к проектированию элемента Cosmos DB больше связан с встраиванием (в Cosmos это называется *вложенностью*), которое можно выполнять с собственными типами, например коллекцией `Tags` в `CosmosBook`. Фактически в документации Cosmos DB (http://mng.bz/EVnq) сказано:

> *Поскольку в настоящее время нет концепции ограничения, внешнего ключа или чего-либо еще, любые связи между документами, которые у вас есть, по сути являются «слабыми ссылками» и не будут проверяться самой базой данных.*

Большинство NoSQL баз данных похожи на Cosmos DB в том, что они не поддерживают связи между элементами. Лично я не уверен, что EF Core должен добавлять связи между разными элементами в базе данных Cosmos, поскольку они не будут работать так, как мы ожидаем, с реляционными базами данных.

16.7 Стоило ли использование Cosmos DB затраченных усилий? Да!

Мы создали систему CQRS с двумя базами данных для повышения производительности и масштабируемости приложения Book App. Кроме того, внедрение системы CQRS с Cosmos DB многому нас научило. Мы узнали, что можно делать с Cosmos, а что нет, а также об ограничениях поставщика EF Core 5. В этом разделе мы рассмотрим три темы:

- производительность системы CQRS с двумя базами данных в приложении Book App;
- функции, с которыми провайдер баз данных EF Core 5 для Cosmos DB не может справиться;
- насколько сложно было бы использовать такой вариант системы CQRS с двумя базами данных в приложении.

Чтобы сравнить производительность и возможности, используются четыре типа запросов:

- *Cosmos* (*EF*) – использует провайдер баз данных EF Core для Cosmos DB;
- *Cosmos* (*Direct*) – использует пакет .NET SDK для Cosmos DB;
- *SQL* (*+ кеш*) – использует кешированные значения в реляционной базе данных (см. раздел 15.5);
- *SQL* (*Dapper*) – использует оптимальный SQL-код для доступа к реляционной базе данных (см. раздел 15.4).

ПРИМЕЧАНИЕ Я опустил исходный код для отображения книг, разработанный в главе 2, потому что он настолько медленный, что это было бы бесполезно. Кроме того, он генерировал исключение для запросов, для которых был превышен тайм-аут базы данных, равный 30 секундам.

Цель состоит в том, чтобы сравнить производительность, функции и затраты на разработку, как я делал это в разделе 15.6, для четырех уровней настройки производительности.

16.7.1 Оценка производительности системы CQRS с двумя базами данных в приложении Book App

Чтобы сравнить производительность подходов из главы 15 и системы CQRS из этой главы, у меня было два типа запросов Cosmos DB с использованием EF Core и напрямую через Cosmos SQL API и два SQL-запроса из главы 15, с SQL (+ *кеш*) и SQL (Dapper). Эти четыре способа получения данных для отображения книг позволили мне сравнить производительность двух типов баз данных.

Чтобы сравнение было справедливым, обе базы данных должны:

- *находиться в одном и том же месте*, чтобы время доступа (latency) было одинаковым. Я добился этого путем создания обеих баз данных на сайте Azure в Лондоне, что составляет около 50 миль от моего местоположения;
- *быть сравнимыми по цене*, потому что цена определяет производительность двух баз данных. Базы данных близки по цене и достаточно дешевы, чтобы их можно было протестировать, не тратя много денег. В табл. 16.1 приведены подробные сведения о двух базах данных.

Таблица 16.1 Две базы данных, используемые для сравнения производительности реляционной базы данных и Cosmos DB

Тип базы данных	Название службы Azure	Единицы производительности	Цена в месяц
Azure SQL Server	Standard	20 DTU	$37
Cosmos DB	Pay as you go	Ручная масштабируемость, 800 единиц запроса	$47

ПРИМЕЧАНИЕ И у Azure SQL Server, и у Cosmos DB есть бессерверная версия, где производительность базы данных может расти и падать в зависимости от спроса. Эта версия могла быть для меня дешевле, но мне нужна была конкретная производительность для сравнения SQL-запросов с запросами Cosmos DB.

В следующем списке показаны наборы сущностей `Book` (SQL-версии `Book` и `CosmosBook`) в базах данных, которые использовались в тестах на производительность. Кроме того, здесь показано количество отзы-

вов в базе данных, поскольку сортировка или фильтрация по средней оценке – одни из самых сложных запросов:

- 100 000 книг, у которых 546 000 отзывов;
- 250 000 книг, у которых 1 365 000 отзывов;
- 500 000 книг, у которых 2 740 000 отзывов.

Моя первая попытка измерить производительность при разных размерах баз данных включала запросы с SQL (+ кеш) и SQL (Dapper), описанные в главе 15. Но оказывается, что производительность при подсчете количества сущностей `Book` в запросе низкая. При 500 000 книг простой просмотр первых 100 книг занял 230 мс. Я чувствовал, что эта оценка Cosmos (EF) и SQL (EF) была несправедливой, поэтому создал версии SQL (+ cacheNC) и SQL (DapperNC). (NC означает *no count*.) Первая диаграмма производительности, где рассматривается производительность по мере роста базы данных, содержащая только Cosmos DB (EF) и Cosmos DB (Direct), показана на рис. 16.7.

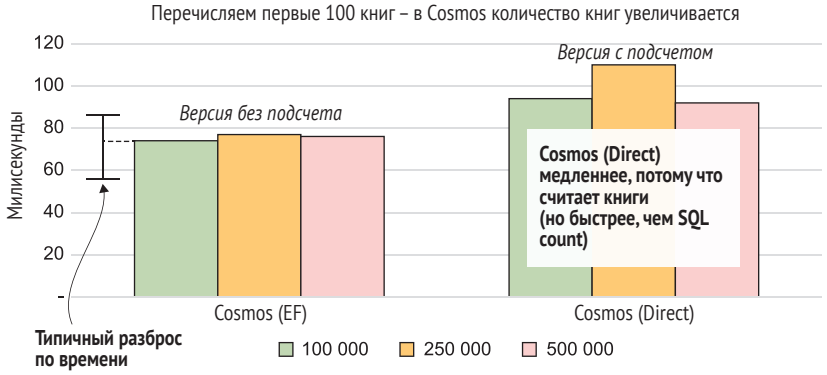


Рис. 16.7 Время, затраченное на отображение HTML-страницы, содержащей первые 100 книг (упорядочены по первичному ключу по убыванию) в контейнере Cosmos DB для трех размеров базы данных. На этом рисунке показано, что размер базы данных мало влияет на время выполнения. Обратите внимание, что этот хронометраж был измерен с разницей в несколько дней, и разница довольно велика (~ 35 мс), поэтому эта диаграмма может выглядеть иначе, если я повторно проведу тест

ПРИМЕЧАНИЕ Весь хронометраж был взят из журнала ASP.NET Core, `RequestFinished`, который содержит общее время выполнения HTTP-запроса. Время запроса измерялось путем выполнения запроса не менее семи раз и взятия среднего значения последних пяти раз. Чтобы получить доступ к этим данным, выберите команду **Admin** > **Timings** в приложении Book App.

Основное различие между Cosmos DB (EF) и Cosmos DB (Direct) заключается в том, что Cosmos DB (Direct) использует оригинальный подход с разбиением на страницы. Это означает, что ей пришлось подсчитать количество книг в общем запросе. На рис. 16.7 показа-

но, что Cosmos DB считает быстро – фактически примерно в два раза быстрее, чем SQL для 500 000 книг. В данном случае скорость не имеет большого значения, но в некоторых приложениях быстрый подсчет может быть очень важен. Следующие тесты на производительность касались большинства ключевых сортировок и фильтров по четырем типам запросов: Cosmos DB (EF), Cosmos DB (Direct), SQL (+ cacheNC) и SQL (DapperNC) для 500 000 книг, как показано на рис. 16.8.

ПРИМЕЧАНИЕ Я обсуждаю эффект извлечения тегов из 500 000 сущностей `CosmosBook` в разделе 16.7.2 во врезке под заголовком «При перегрузке базы данных Cosmos DB происходят интересные вещи».

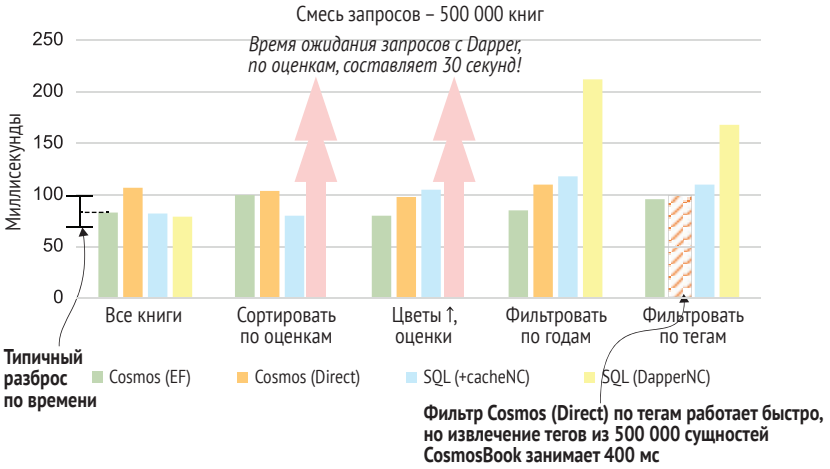


Рис. 16.8 Время, затраченное на пять ключевых запросов к базам данных, содержащим 500 000 книг. Четыре типа доступа к базе данных: Cosmos DB (EF), Cosmos DB (Direct), SQL (+ cacheNC) и SQL (DapperNC)

На рис. 16.8 показана информация, чтобы сделать выводы, которые приведены в следующем списке. Важные факты перечислены вначале:

- даже лучшая SQL-версия (DapperNC) не работает в этом приложении, потому что любая сортировка или фильтрация отзывов занимала так много времени, что превышался 30-секундный тайм-аут базы данных;
- версия SQL (+ cacheNC) была на уровне или лучше с Cosmos DB (EF) в первых двух запросах, но по мере того, как запрос становился более сложным, производительность стала отставать;
- Cosmos DB (Direct) был примерно на 25 % медленнее, чем Cosmos DB (EF), без подсчета количества, но все равно он примерно в два раза быстрее, чем SQL с подсчетом.

В целом я думаю, что этот тест показывает победу Cosmos DB, особенно если учесть тот факт, что реализация данной системы CQRS

была проще и быстрее, чем создание исходной версии SQL (+ кеш). Кроме того, обработка параллелизма в Cosmos DB (см. раздел 16.4.5) проще, чем в SQL-версии.

Конечно, у подхода CQRS / Cosmos DB есть и недостатки. Во-первых, добавление и обновление книги занимает немного больше времени, потому что CQRS требуется четыре обращения к базе данных: два для обновления реляционной базы данных и два для обновления базы данных Cosmos. Эти обновления в сумме составляют около 110 мс, что более чем вдвое превышает время, необходимое только для реляционной базы данных. Поэтому если приложение выполняет много операций записи в базу данных, то этот подход, возможно, вам не подойдет.

> **ПРИМЕЧАНИЕ** Есть несколько способов улучшить производительность записи при использовании CQRS за счет более сложного кода. Некоторые из них описаны в одной из моих статей: http://mng.bz/N8dE.

Второй недостаток – особенность Cosmos DB: использовать LINQ-метод `Skip` долго и затратно (см. раздел 16.6.4). На рис. 16.9 показано, что чем больше книг «обрабатывается» с помощью этого метода, тем больше времени занимает процесс. Время не должно быть проблемой для приложения Book App, поскольку многие сдались бы после нескольких страниц, но если приложению требуется глубокая «обработка» данных с помощью метода `Skip`, то Cosmos DB для этой цели не подходит.

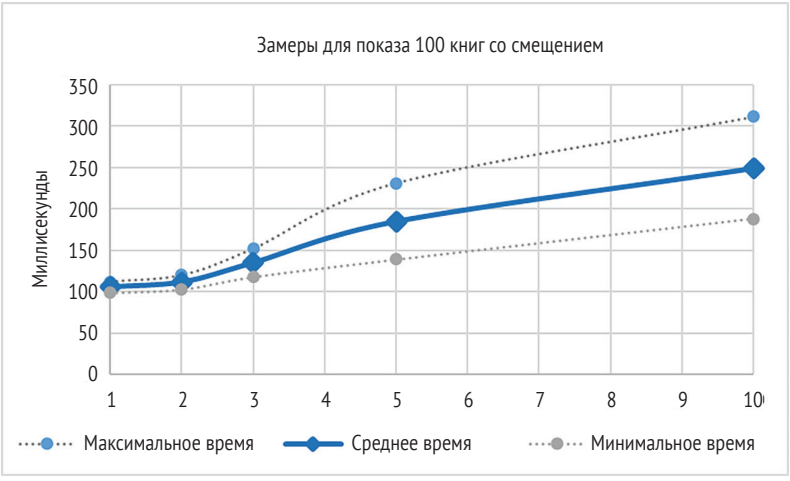


Рис. 16.9 Чем больше книг «обрабатывается» с помощью метода `Skip`, тем больше увеличивается время выполнения. На диаграмме показаны максимальный и минимальный диапазоны хронометража пяти замеров. Видно, чем больше элементов было пропущено, тем больше увеличивается разброс

16.7.2 Исправление функций, с которыми поставщик баз данных EF Core 5 для Cosmos DB не справился

При создании исходной реализации запросов к Cosmos DB через провайдера баз данных EF Core 5 я ограничился использованием только тех функций, которые были предоставлены EF Core 5. Но когда вы создаете настоящее приложение, вы используете то, что у вас есть, а затем импровизируете, потому что вам платят за то, чтобы приложение работало. В этом разделе мы исправим проблемы, уже упомянутые в этой главе:

- не удалось быстро подсчитать количество книг в Cosmos DB;
- не удалось создать фильтр «По годам публикации» в разумные сроки;
- не удалось создать фильтр «По тегам»;
- не удалось выполнить фильтрацию по TagId, поскольку провайдер EF Core 5 не поддерживает команду `IN`.

ПРИМЕЧАНИЕ Этот раздел касается только обработки ограничений в провайдере баз данных EF Core 5 для Cosmos DB. Он не распространяется на ограничения Cosmos DB, такие как необходимость разбивать сложные запросы (см. раздел 16.6.3).

НЕ УДАЛОСЬ БЫСТРО ПОДСЧИТАТЬ КОЛИЧЕСТВО КНИГ В COSMOS DB

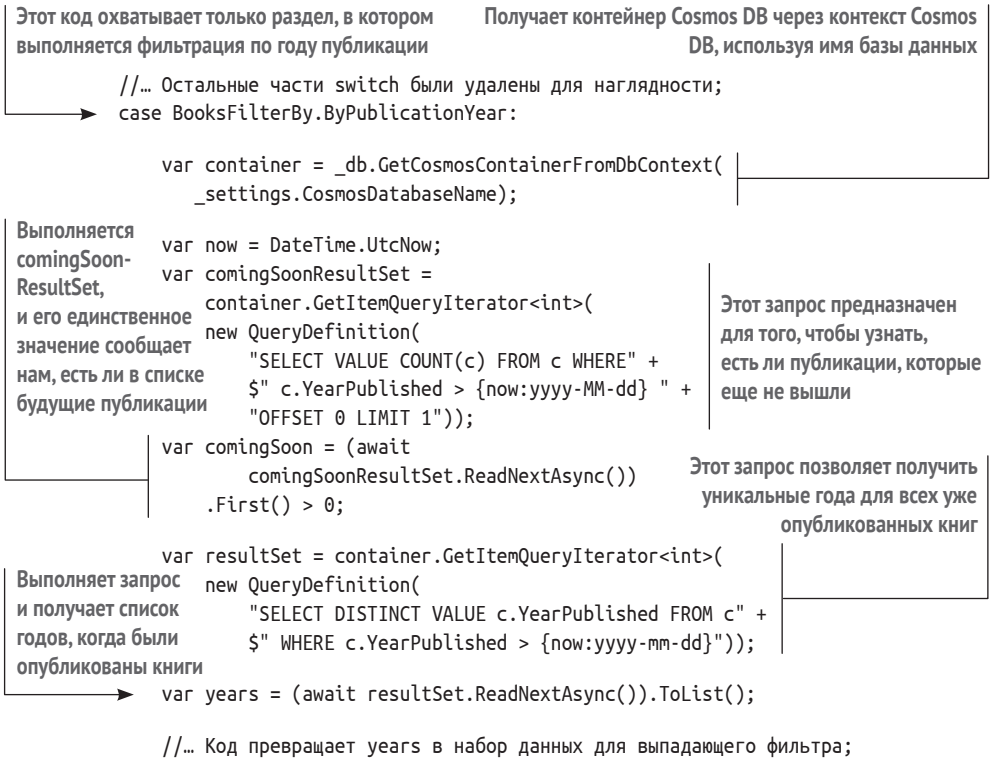
Этот вопрос рассматривается в разделе 16.6.8. Переход к варианту разбиения на страницы `Next/Previous` вполне приемлем во многих случаях. Этот подход используется Amazon, так что он должен сработать для сайтов, продающих книги.

НЕ УДАЛОСЬ СОЗДАТЬ ФИЛЬТР «ПО ГОДАМ ПУБЛИКАЦИИ» В РАЗУМНЫЕ СРОКИ

При выборе фильтра «По годам публикации» код должен просмотреть все книги, чтобы найти свойство `YearPublished` и использовать метод `Distinct`, дабы получить все уникальные года. Проблема здесь состоит не в том, что LINQ-запрос нельзя выполнить, а в том, что он довольно медленный (25 секунд на 500 000 книг). Подозреваю, что он был медленным, потому что метод `Distinct` выполнялся на стороне приложения, однако не уверен.

Но я знаю, что использование прямых SQL-команд Cosmos может сработать. Фактически такая команда быстрее, чем SQL-версия. На 500 000 книг у Cosmos SQL ушло ~ 400 мс, тогда как у SQL – ~ 2,5 секунды. Итак, решено – использовать прямую SQL-команду Cosmos для получения списка годов. Для этого требуется получить контейнер Cosmos DB через контекст Cosmos DB, как показано в следующем листинге.

Листинг 16.11 Сервис `Filter Drop-down`, показывающий использование прямой Cosmos SQL-команды




```
//… Остальные части switch были удалены для наглядности;
case BooksFilterBy.ByPublicationYear:

    var container = _db.GetCosmosContainerFromDbContext(
       _settings.CosmosDatabaseName);

    var now = DateTime.UtcNow;
    var comingSoonResultSet =
        container.GetItemQueryIterator<int>(
        new QueryDefinition(
            "SELECT VALUE COUNT(c) FROM c WHERE" +
            $" c.YearPublished > {now:yyyy-MM-dd} " +
            "OFFSET 0 LIMIT 1"));
    var comingSoon = (await
            comingSoonResultSet.ReadNextAsync())
        .First() > 0;

    var resultSet = container.GetItemQueryIterator<int>(
        new QueryDefinition(
            "SELECT DISTINCT VALUE c.YearPublished FROM c" +
            $" WHERE c.YearPublished > {now:yyyy-mm-dd}"));
    var years = (await resultSet.ReadNextAsync()).ToList();

    //… Код превращает years в набор данных для выпадающего фильтра;
```

Но будьте осторожны: запрос, показанный в листинге 16.11, – это еще один запрос с большим количеством единиц запроса, примерно столько же, сколько и у `TagId`, 2321 единица. В этом случае может подойти использование статического списка, поскольку технические книги старше пяти лет обычно бесполезны (за исключением, конечно же, книги «Предметно-ориентированное проектирование» Эрика Эванса!).

НЕ УДАЛОСЬ СОЗДАТЬ ФИЛЬТР «ПО ТЕГАМ»

EF Core 5 не удалось получить отдельный набор TagId из коллекции Tags каждой сущности `CosmosBook`, потому что провайдер базы данных EF Core 5 для Cosmos не поддерживает подзапросы. Опять же, вместо этого можно использовать прямые SQL-команды Cosmos – Cosmos (Direct) занимает всего ~ 350 мс, но это затратно. Получить список `TagId` из реляционной базы данных просто, потому что в ней есть таблица с именем Tags, содержащая всего 35 строк. Поэтому, вместо того чтобы просматривать 500 000 сущностей `CosmosBook` и извлекать все TagId, можно просто выполнить следующий SQL-код, который занимает всего ~ 30 мс:

```
var drop-down = _sqlContext.Tags
    .Select(x => new DropDownTuple
    {
        Value = x.TagId,
        Text = x.TagId
    }).ToList();
```

При перегрузке базы данных Cosmos DB происходят интересные вещи

При создании Cosmos (Direct) для фильтрации по тегам я решил извлечь `TagId` с помощью SQL-команды Cosmos:

```
SELECT DISTINCT VALUE f.TagId FROM c JOIN f in c.Tags
```

Эта команда работает, но занимает много времени (~ 400 мс) и стоит дорого – 2445 единиц запроса, если быть точным. Поскольку эта команда превышает 800 единиц, выделенных для моей базы данных (контейнер Cosmos), Cosmos штрафует за любые запросы, превышающие лимит.

В этом случае Cosmos, казалось, замедлилась на несколько секунд, и с меня списали больше за превышение 800 единиц, за которые я заплатил. Постарайтесь сохранять стоимость своих запросов в рамках выделенных средств, если не хотите, чтобы последующие запросы выполнялись медленно.

Должен сказать, что просьба Cosmos DB извлечь все `TagId` из всех 500 000 сущностей `CosmosBook` для получения 35 уникальных TagId – не лучший подход, но он показал, что произойдет, если превысить выделенные единицы запроса.

НЕ УДАЛОСЬ ВЫПОЛНИТЬ ФИЛЬТРАЦИЮ ПО TagId, ПОТОМУ ЧТО ПРОВАЙДЕР EF CORE 5 НЕ ПОДДЕРЖИВАЕТ КОМАНДУ IN

Последняя проблема, которую необходимо решить, – это фильтрация книг по `TagId`, потому что провайдер базы данных EF Core 5 для Cosmos DB не поддерживает команду `IN`. Хотя можно было бы использовать прямую SQL-команду Cosmos, EF Core 5 поддерживает метод LINQ `Contains` для строк.

ОГРАНИЧЕНИЕ EF CORE 5 EF Core 5 не поддерживает эквивалент Cosmos для SQL-команды `IN`, чтобы выполнять фильтрацию по коллекции Tags. Запрос LINQ `Books.Where(x => x.Tags.Any(y => y == "имя тега"))` выбросит исключение `could not be translated`. Я обхожу эту проблему, используя метод `Contains`.

Добавив строку с именем `TagsString` и вставив каждый `TagId`, а также добавив дополнительные разделители символов, можно использовать этот метод для фильтрации по `TagId`. Следующий фрагмент кода

показывает пару «ключ/значение» `TagsString`, взятую из данных `CosmosBook`, хранящихся в формате JSON из листинга 16.10:

```
"TagsString": "| Databases | Microsoft & .NET |"
```

> **ПРИМЕЧАНИЕ** Символ разделения | в `TagsString` гарантирует, что фильтрация по тегам будет соответствовать строке из `TagId`; в противном случае тег `"Data"` совпадал бы с `"Data"` и `"Databases"`.

Эта техника упрощает фильтрацию по тегу. Например, чтобы выбрать все книги с тегом `"C#"`, можно было бы написать

```
context.Books
    .Where(x => x.TagsString.Contains("| C# |"))
    .ToListAsync();
```

Такой подход вполне приемлем в Cosmos DB, где есть специальный API для `Contains` и строк. Фактически строковый метод `Contains` работает быстрее, чем метод вложенного запроса `IN`. На 500 000 книг у `Contains` ушло ~ 125 мс, тогда как у версии `JOIN/WHERE` был большой разброс по хронометражу, до 3 секунд.

16.7.3 Насколько сложно было бы использовать эту систему CQRS с двумя базами данных в своем приложении?

Нет сомнений в том, что версия с Cosmos DB обеспечивает отличную производительность приложения Book App при увеличении количества книг и отзывов. Но насколько сложно было бы добавить этот подход к существующему приложению, и будет ли это иметь отрицательные последствия для разработки приложения в дальнейшем? Я добавил данный вариант с CQRS в существующее приложение Book App, поэтому могу ответить на эти вопросы.

Поразмыслив, я понял, что бóльшая часть времени уходит на понимание того, как работает Cosmos DB, и на настройку в соответствии с ее стилем. Глядя на коммиты в GitHub, мне потребовалось около двух недель, чтобы добавить улучшение для CQRS с двумя базами данных в существующее приложение Book App, но это время включало в себя большое количество исследований и создание дополнительной версии Cosmos (Direct). Как я сказал ранее, я думаю, что CQRS с двумя базами данных немного проще создать и протестировать, чем версию SQL (+ кеш).

> **ПРИМЕЧАНИЕ** Усовершенствование версии CQRS с двумя базами данных было реализовано как дополнительный подход к выполнению запросов, не учитывая при этом все исходные системы отображения книг; кроме того, я изменил код версий *SQL* (*+ кеш*) и *SQL* (*Dapper*), чтобы получить версию без подсчета

количества. Сборка всех этих версий позволила мне сравнить производительность системы CQRS с двумя базами данных с оригинальными реляционными системами отображения книг.

Я разбил свои взгляды на то, насколько сложно все это было, на части:

- *обнаружение изменений в SQL-версии сущности* `Book` – эта часть была упрощена благодаря использованию DDD-классов, поскольку я мог добавить событие к каждому методу доступа в классе сущности `Book`. Если вы не используете эти классы, вам нужно будет обнаруживать изменения в сущностях, используя метод `SaveChangesAsync`, но, как я уже сказал в разделе 16.4, этот подход сложнее;
- *запуск кода события в транзакции* – моя библиотека `GenericEventRunner` значительно ускорила написание этой части. Вам не обязательно использовать эту библиотеку, но тогда на разработку уйдет больше времени;
- *запись в базу данных Cosmos DB* – эта часть была довольно простой, здесь использовались простые методы `Add`, `Update` и `Delete`. (См., например, листинг 16.8.) Я потратил некоторое время на то, чтобы сделать операцию записи более устойчивой, устраняя возможные причины посредством одновременных обновлений;
- *запросы к базе данных Cosmos DB* – эта часть заняла больше всего времени, в основном потому, что в EF Core и Cosmos DB есть ограничения.

Когда дело дошло до добавления CQRS к существующему приложению Book App, я бы сказал, что часть, касающаяся Cosmos DB, мало повлияла на структуру приложения. Далее приводятся изменения, которые мне нужно было внести в существующий код:

- регистрация Cosmos DbContext при запуске;
- добавление событий интеграции в класс сущности `Book`;
- изменение кода *SQL* (*+ кеш*) и *SQL* (*Dapper*) для создания версий без подсчета количества.

Весь существующий код по-прежнему работает так же, как и всегда. Очевидно, что изменения в сущности `Book` могут потребовать изменений сущности `CosmosBook` и связанного с ней метода расширения `MapBookToCosmosBook`. За исключением изменений сущности `Book`, изменение в коде SQL не должно влиять на код Cosmos DB, а изменение кода Cosmos DB не должно влиять на SQL-код приложения.

16.8 Отличия в других типах баз данных

Большая часть этой главы посвящена Cosmos DB, которая отличается от реляционных баз данных, о которых идет речь в этой книге. Но в конце главы мы снова рассмотрим реляционные базы данных. Раз-

личные типы реляционных баз данных похожи в основном потому, что существует официальный стандарт языка SQL, но есть и множество небольших отличий. Этот раздел может оказаться полезным, если вы хотите перейти с одной реляционной базы данных на другую, например поменять SQL Server на PostgreSQL.

EF Core будет обрабатывать многие различия между типами реляционных баз данных, например то, как имена таблиц должны быть включены в команды SQL, но что-то придется делать самим, например когда речь идет о различных форматах пользовательских функций (см. раздел 10.1). Вот список типичных вещей, которые нужно проверить и изменить, если вы переходите с одной реляционной базы данных на другую:

1. Скачайте провайдер базы данных в NuGet и измените регистрацию DbContext.
   Первое, что нужно сделать, – это установить конкретного провайдера базы данных EF Core через NuGet, например `Microsoft.EntityFrameworkCore.SqlServer` или `Npgsql .EntityFrameworkCore.PostgreSQL`. После этого нужно изменить способ регистрации этого провайдера базы данных у себя в DbContext. В ASP.NET Core для провайдера базы данных MySQL у вас будет что-то вроде этого:

   ```
   services.AddDbContext<MyDbContext>(
       options => options.UseMySql(connection));
   ```

2. Повторно выполните команду `Add-Migration` для нового провайдера базы данных.
   Миграции EF Core зависят от провайдера базы данных и *не* подлежат передаче от одной базы данных к другой. Нужно удалить старые миграции и выполнить команду `Add-Migration`, используя нового провайдера базы данных.

ПРИМЕЧАНИЕ У вас могут быть миграции для нескольких типов баз данных, только при условии, что они хранятся в разных проектах. Нужно добавить метод `MigrationsAssembly` к регистрации каждого DbContext, чтобы сообщить EF Core, где расположены миграции.

3. Исправьте любое отображение типов между .NET и базой данных, которое изменилось.
   Нужно повторно выполнить LINQ-запросы и посмотреть, изменилось ли что-нибудь. В первом издании этой книги я преобразовал приложение Book App, заменив SQL Server на MySQL, и основной запрос для отображения книг с методом `Select` (см. листинг 2.12) выбросил исключение. Оказывается, возвращаемый тип SQL-команды `AVG` в MySQL – это `decimal`, допускающий значение NULL, а не `double`, допускающий значение NULL, в SQL

Server. Чтобы преодолеть эту проблему, нужно изменить тип свойства `AverageReviewVotes` у `BookListDto` на `decimal`, чтобы соответствовать тому, как работает MySQL.

Существуют и другие, более тонкие различия типов между серверами баз данных, которые могут остаться незамеченными. Вот типичные вещи, на которые стоит обратить внимание:

a *типы временных меток параллелизма* – в SQL Server это тип `byte[]`; в PostgreSQL используется тип `uint` (и нужно настроить его, когда вы регистрируете свой DbContext); а MySQL использует тип `DateTime`, поэтому убедитесь, что в используемой вами базе данных поддерживается правильный тип;

b *строковые запросы и сопоставление* (см. раздел 2.8.3) – по умолчанию SQL Server и MySQL используют совпадение строк без учета регистра, а PostgreSQL – по умолчанию с учетом регистра. При настройке сопоставления для базы данных, таблицы или столбца используются разные названия, и эффект будет разным;

c *точность* `DateTime` – большинство баз данных перешли на `DateTime2` с точностью до 100 нс, но лучше проверить. SQLite хранит `DateTime` в виде строки в формате ISO8601: `"YYYY-MM-DD HH:MM:SS.SSS"`.

4 Проверьте и измените любой чистый SQL-код, который вы используете.

На этом этапе все становится еще сложнее, потому что EF Core не охватывает никаких изменений относительно того, как тип базы данных использует SQL. Стандартный код SQL должен работать, но способ обращения к таблицам и столбцам может измениться. Более сложный SQL, например пользовательские функции и хранимые процедуры, похоже, имеет несколько разные форматы для разных типов баз данных.

Резюме

- NoSQL база данных предназначена для обеспечения высокой производительности с точки зрения скорости, масштабируемости и доступности. Эта производительность достигается за счет отказа от некоторых возможностей реляционных баз данных, таких как сильные связи между таблицами.
- Архитектура CQRS отделяет операции чтения от операций записи, что позволяет улучшить производительность чтения данных, сохраняя данные в форме, соответствующей запросу, известной как проекция.
- Приложение Book App было дополнено возможностью хранить проекцию SQL-версии сущности `Book` на стороне чтения архитектуры CQRS, которая использует базу данных Cosmos DB. Такой подход

улучшает производительность, особенно при большом количестве записей.

- Дизайн, используемый для реализации архитектуры SQL / Cosmos DB CQRS, использует событие интеграции (см. главу 12).
- База данных Cosmos DB работает иначе, чем реляционные базы данных, и процесс добавления ее в приложение Book App обнажает множество этих различий.
- У провайдера базы данных EF Core 5 для Cosmos DB есть много ограничений, которые обсуждаются и преодолеваются в этой главе. Но разработать полезное приложение с Cosmos DB все же можно.
- Обновленное приложение Book App показывает, что база данных Cosmos DB может обеспечить лучшую производительность операций чтения по сравнению с базой данных SQL Server с аналогичной ценой.
- Дизайн SQL / Cosmos DB CQRS подходит для добавления в существующее приложение, где требуется повышение производительности на стороне чтения, но при этом добавляются временные затраты на каждое добавление или обновление данных.
- Реляционные базы данных больше похожи друг на друга, чем на NoSQL базы данных, из-за стандартизации языка SQL. Но нужно внести некоторые изменения и выполнить проверки при переходе с одного типа реляционной базы данных на другой.

17 Модульное тестирование приложений, использующих EF Core

В этой главе рассматриваются следующие темы:

- моделирование базы данных для модульного тестирования;
- использование для модульного тестирования того же типа базы данных, что и в рабочем приложении;
- использование SQLite in-memory базы данных для модульного тестирования;
- решение проблемы доступа к одной базе данных, нарушающей другую часть теста;
- сбор данных журнала во время модульного тестирования.

Эта глава посвящена модульному тестированию приложений, использующих EF Core для доступа к базе данных. Вы узнаете, какие подходы к модульному тестированию доступны для работы с EF Core и как выбрать инструменты, подходящие для ваших нужд. Кроме того, я описываю многочисленные методы и приемы, которые сделают ваше модульное тестирование всесторонним и эффективным. Лично я считаю полезной практику модульного тестирования и часто ею пользуюсь, поскольку она позволяет отлавливать ошибки не толь-

ко в момент написания кода, но и, что еще важнее, во время его рефакторинга, улучшая тем самым качество разработки.

Хотя мне очень нравится модульное тестирование, я осознаю, что написание модульных тестов требует усилий со стороны разработчиков, включая их рефакторинг по мере роста приложения. За годы работы я изучил множество рекомендаций и приемов по написанию модульных тестов и создал библиотеку под названием EfCore.TestSupport, которая поможет вам быстро и эффективно писать модульные тесты.

Модульное тестирование – большая тема, которой посвящены целые книги. Я сосредоточусь на узкой, но важной области модульного тестирования приложений, использующих EF Core для доступа к базе данных. Чтобы не отходить далеко от темы, заявленной в главе, я не стану объяснять основы модульного тестирования, а сразу перейду к делу. Если же вы новичок в модульном тестировании, я советую вам пропустить данную главу и вернуться к ней снова, после того как вы его освоите. Эта глава не будет иметь смысла без данного навыка, и я не хочу разочаровывать вас в модульном тестировании, слишком усложняя его.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ Чтобы получить представление о модульном тестировании в .NET, посмотрите это видео: http://mng.bz/K44E. Для более глубокого изучения модульного тестирования рекомендую книгу Владимира Хорикова «*Принципы юнит-тестирования*» (https://www.manning.com/books/unit-testing).

Отлично, если вы все еще здесь, то я предполагаю, вы знаете, что такое модульное тестирование, и написали хотя бы несколько модульных тестов. Я не буду описывать различия между модульными и интеграционными тестами, приемочными тестами и т. д. Я здесь не для того, чтобы убеждать вас в полезности модульных тестов; предполагаю, что вы убеждены в их полезности и хотите изучить приемы и методы модульного тестирования приложения, использующего EF Core.

ПРИМЕЧАНИЕ Тесты, которые используют реальную базу данных, некоторые люди называют *интеграционными*. Я же называю все свои тесты *модульными*.

Как я уже сказал, я часто использую модульные тесты. В репозитории на GitHub для этой книги более 700 модульных тестов, некоторые для проверки работоспособности моего приложения Book App, а некоторые – для проверки правильности того, о чем я говорю в книге. Эти тесты вселяют в меня уверенность в том, что в книге все написано правильно и что приложение Book App работает правильно. Часть кода в версии этого приложения для третьей части довольно сложная, и именно здесь модульные тесты становятся наиболее полезными.

ПРИМЕЧАНИЕ Артур Викерс, технический руководитель EF Core, написал в Twitter, что в EF Core более 70 000 модульных тестов (с использованием xUnit). См. http://mng.bz/D18y.

Еще одна вещь, которую я усвоил, – я хочу, чтобы мои модульные тесты выполнялись как можно быстрее, потому что быстрый цикл тестирования-отладки делает разработку и рефакторинг приложения намного приятнее. Кроме того, у меня гораздо больше шансов запустить все свои модульные тесты, если они будут быстрыми, что поможет выявить ошибки в местах, которые, как мне казалось, не будут затронуты моим новым кодом. Я обобщил эти два аспекта модульного тестирования на рис. 17.1.

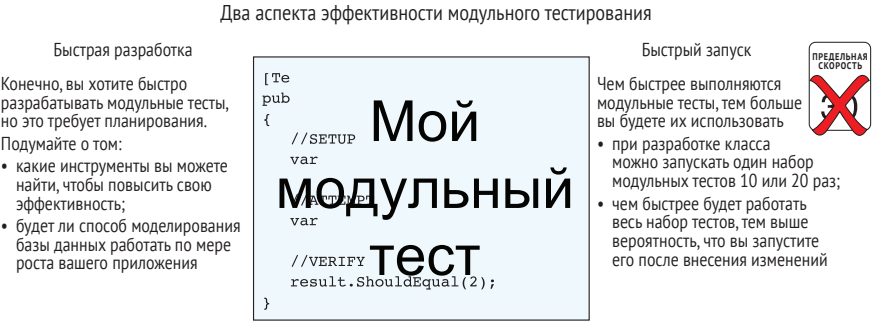


Рис. 17.1 Я искренне верю в модульные тесты, но это не значит, что я хочу тратить много времени на их разработку и выполнение. Мой подход состоит в том, чтобы попытаться эффективно использовать их, что включает в себя быструю разработку и отсутствие необходимости бездельничать во время их выполнения

Остальная часть главы начинается с основ, здесь же рассматриваются способы написания модульных тестов и, наконец, представлены конкретные советы и проблемы, которые могут возникнуть при тестировании кода EF Core. Вот разделы этой главы:

- знакомство с настройкой модульного теста;
- подготовка DbContext приложения к модульному тестированию;
- три способа моделирования базы данных при тестировании приложений EF Core:
  - использование базы данных промышленного типа в модульных тестах;
  - использование SQLite in-memory базы данных для модульного тестирования;
  - создание заглушки или имитации базы данных EF Core;
- модульное тестирование базы данных Cosmos DB;
- заполнение базы данных тестовыми данными для правильного тестирования кода;
- решение проблемы доступа к одной базе данных, нарушающей другую часть теста;
- перехват команд базы данных, отправляемых в базу данных.

17.1 Знакомство с настройкой модульного теста

Прежде чем я начну объяснять методы, мне нужно познакомить вас с настройкой нашего модульного теста; в противном случае примеры будут лишены смысла. Я использую довольно стандартный подход, но, как вы увидите, я также создал инструменты, которые помогут вам с EF Core и модульным тестированием при использовании базы данных. На рис. 17.2 показан модульный тест, в котором используются некоторые рассмотренные в этой главе функциональные возможности и методы.

ПРИМЕЧАНИЕ Все модульные тесты в этой главе (кроме раздела Cosmos DB, 17.8) используют синхронные методы; например, они вызывают метод `SaveChanges`, а не `SaveChangesAsync`. Отчасти я делаю это потому, что без ключевого слова `await` код немного легче понять, но в реальной жизни я использую синхронные методы всякий раз, когда есть такая возможность, потому что такой код обеспечивает более понятный вывод `StackTrace` при возникновении исключения и его проще отлаживать, используя точки останова.

Темы этого раздела:

- тестовое окружение, которое вы будете использовать: библиотека модульного тестирования xUnit;
- пакет NuGet, который я создал, чтобы было проще проводить модульное тестирование приложений, использующих EF Core.

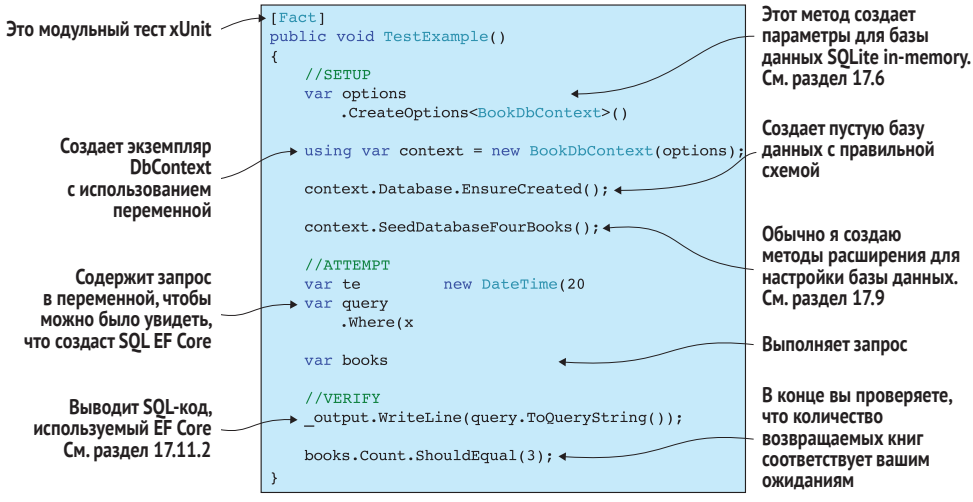


Рис. 17.2 Модульный тест, состоящий из трех частей: настройка, действие и проверка (также известные как паттерн AAA (Arrange, Act and Assert)). Кроме того, на рисунке показаны методы EF Core, которые будут объяснены в этой главе

17.1.1 Окружение тестирования: библиотека модульного тестирования xUnit

Я использую библиотеку модульных тестов xUnit (https://xunit.net), потому что компания Microsoft должным образом ее поддерживает и ее использует команда EF Core. Кроме того, xUnit работает быстрее по сравнению с некоторыми другими библиотеками модульного тестирования, например NUnit (которую я когда-то использовал), потому что xUnit может запускать классы модульного тестирования параллельно. У параллельного запуска тестов есть и обратная сторона. Я покажу вам, как ее обойти, но это означает, что вы можете выполнить полный набор модульных тестов намного быстрее.

Кроме того, я применяю *Fluent-валидацию, которая использует ряд методов расширения, следующих один за другим*; см. строку 1 в табл. 17.1. Я считаю, что с таким стилем намного проще работать, чем со статическими методами `Assert`; он немного короче, и Intellisense может предложить подходящие методы.

Таблица 17.1 Два подхода к проверке того, что две книги были загружены предыдущим тестируемым запросом. Статические методы `Assert` встроены в XUnit; в качестве дополнительного шага нужно добавить стиль Fluent-валидации

Тип	Пример кода
Стиль Fluent-валидации	`books.Count().ShouldEqual(2);`
Статический метод `Assert`	`Assert.Equal(2, books.Count());`

Эти методы расширения для Fluent-валидации можно найти на странице http://mng.bz/l2Ej, но можно создать и собственные; это обычные методы расширения C#. Я включил примеры расширенных методов Fluent-валидации xUnit, а также несколько дополнительных валидаций в созданный мною пакет NuGet под названием EfCore. TestSupport. Смотрите раздел 17.1.2.

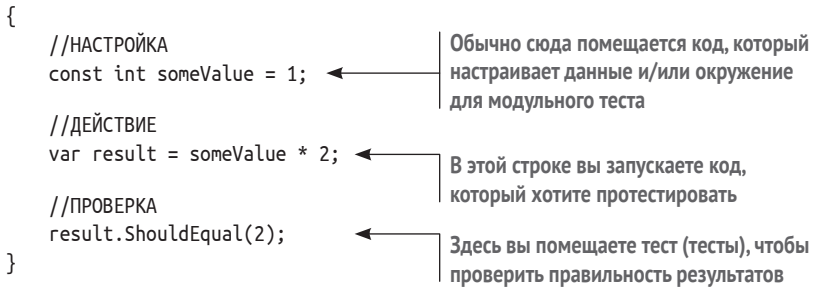
В следующем листинге показан простой модульный тест, в котором используется пакет модульного тестирования xUnit и расширения Fluent-валидации. В этом примере используется трехэтапный паттерн «Настройка, действие и проверка». В модульных тестах в этой главе он показан как `//НАСТРОЙКА`, `//ДЕЙСТВИЕ` и `//ПРОВЕРКА`. Этот паттерн также известен как *Arrange, Act and Assert*, но поскольку я страдаю дислексией, то предпочитаю использовать в моем коде вариант `//НАСТРОЙКА`, `//ДЕЙСТВИЕ` и `//ПРОВЕРКА`, потому что они выглядят совершенно поразному.

Листинг 17.1 Простой пример метода модульного теста xUnit

```
[Fact]
public void DemoTest()
```

Атрибут [Fact] сообщает средству запуска модульного теста, что этот метод – модульный тест xUnit, который следует запустить

Метод должен быть иметь модификатор доступа public.
Он должен возвращать void или, если вы используете асинхронные методы, Task



Можно запускать модульные тесты с помощью встроенного в Visual Studio Обозревателя тестов, доступ к которому осуществляется из меню «Тест». Если вы используете Visual Studio Code (VS Code), средство запуска тестов здесь также встроено, но необходимо настроить задачи `build` и `test` в файле VS Code, tasks.json, который позволяет запускать все тесты с помощью команды **Task** > **Test**.

17.1.2 Созданная мной библиотека для модульного тестирования приложений, использующих EF Core

Я много узнал о модульном тестировании приложений, использующих EF Core, по мере создания программного обеспечения для первого издания этой книги. В результате я создал библиотеку с открытым исходным кодом под названием EfCore.TestSupport (https://github.com/JonPSmith/EfCore.TestSupport), содержащую множество методов, которые можно использовать на этапе настройки модульного теста.

Библиотека EfCore.TestSupport различает EF Core 2 и EF Core 3, используя netstandard, который использовали они, но теперь после выхода EF Core 5 эти различия больше не актуальны. Поэтому я приравнял версию этой библиотеки к EF Core, используя первую часть номера версии. Например, для EF Core 5 потребуется EfCore.TestSupport версии 5.

> **ПРИМЕЧАНИЕ** Читателям, которые уже используют мою библиотеку EfCore.TestSupport, следует иметь в виду, что я также воспользовался возможностью, чтобы привести в порядок библиотеку EfCore.TestSupport, в которую внесены критические изменения. В SqliteInMemory есть изменения (см. раздел 17.6), некоторые методы теперь устарели, и я переместил код EfSchemaCompare в другую библиотеку. Посетите страницу http://mng.bz/BK5v для получения более подробной информации.

В этой главе используются многие методы из библиотеки EfCore.TestSupport, но я не описываю их сигнатуры, потому что на странице http://mng.bz/dmND содержится документация к данной библиотеке. Однако я объясню, как и зачем проводить модульное тестирование, используя некоторые методы из этой библиотеки, и покажу часть кода, который я тоже разработал.

17.2 Подготовка DbContext приложения к модульному тестированию

Прежде чем вы сможете провести модульное тестирование DbContext приложения с базой данных, необходимо убедиться, что вы можете изменить строку подключения к базе данных. В противном случае вы не сможете предоставить другую базу (базы) данных для модульного тестирования. Используемая техника зависит от того, как DbContext приложения принимает настройки параметров. Вот два подхода, которые предоставляет EF Core для настройки параметров:

- DbContext приложения ожидает, что параметры будут переданы в конструктор. Этот подход рекомендуется для приложений ASP.NET Core и .NET Generic.Host;
- DbContext приложения настраивает параметры в методе `OnConfiguring`. Этот подход рекомендуется для приложений без внедрения зависимостей.

17.2.1 Параметры DbContext приложения передаются в конструктор

Если параметры передаются в конструктор DbContext, не нужно вносить какие-либо изменения в DbContext для работы с модульным тестом. У вас уже есть полный контроль над параметрами, предоставленными конструктору DbContext; вы можете изменить строку подключения к базе данных, тип используемого провайдера базы данных и т. д. В следующем листинге показан формат DbContext, в котором используется конструктор для получения параметров. Конструктор выделен жирным шрифтом.

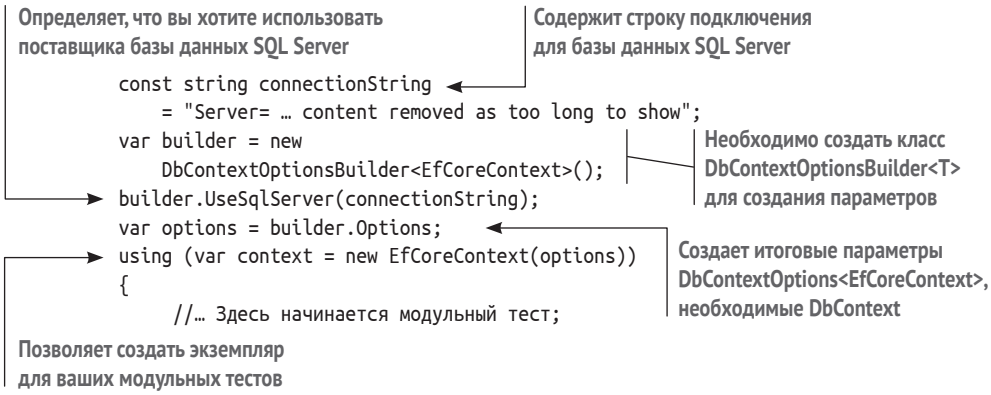
Листинг 17.2 DbContext приложения, использующий конструктор для настройки параметров

```
public class EfCoreContext : DbContext
{
    public EfCoreContext(
        DbContextOptions<EfCoreContext> options)
        : base(options) {}

    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    //… Остальная часть класса не указана;
}
```

Для этого типа DbContext модульный тест может создать переменную options и указать это значение в качестве параметра конструктора. В следующем листинге показан пример создания экземпляра

DbContext в модульном тесте, который будет обращаться к базе данных SQL Server с определенной строкой подключения.

Листинг 17.3 Создание DbContext путем предоставления параметров через конструктор



17.2.2 Настройка параметров DbContext приложения через OnConfiguring

Если параметры базы данных заданы в методе `OnConfiguring` класса DbContext, вы должны изменить DbContext, прежде чем сможете использовать его в модульном тестировании. Но, прежде чем изменить его, хочу показать вам обычную схему использования метода `OnConfiguring` для настройки параметров.

Листинг 17.4 DbContext, использующий метод `OnConfiguring` для настройки параметров

```
public class DbContextOnConfiguring : DbContext
{
    private const string connectionString
        = "Server=(localdb)\\... shortened to fit";
    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
        base.OnConfiguring(optionsBuilder);
    }
    // … Остальной код удален;
}
```

В следующем листинге показан рекомендуемый Microsoft способ изменения DbContext, в котором для настройки параметров используется метод `OnConfiguring`. Как вы увидите, эта техника добавляет тот же тип настройки конструктора, который использует ASP.NET Core,

чтобы сохранить корректную работу метода `OnConfiguring` в обычном приложении.

Листинг 17.5 Измененный DbContext, позволяющий настроить строку подключения в модульном тесте

```
public class DbContextOnConfiguring : DbContext
{
    private const string ConnectionString
        = "Server=(localdb)\\ … shortened to fit";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder
                .UseSqlServer(ConnectionString);
        }
    }

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

    public DbContextOnConfiguring() { }
    // … Остальной код удален;
}
```

Изменяет метод OnConfigured для запуска его обычного кода настройки, только если параметры еще не настроены

Добавляет те же настройки параметров в конструкторе, что и версия ASP.NET Core. Это позволяет устанавливать любые необходимые параметры

Добавляет открытый конструктор без параметров, чтобы этот DbContext корректно работал в приложении

Чтобы использовать эту измененную форму, можно предоставить параметры так же, как и в версии с ASP.NET Core, как показано в следующем листинге.

Листинг 17.6 Модульный тест, предоставляющий другую строку подключения к DbContext

Содержит строку подключения для базы данных, которая будет использоваться для модульного теста

```
const string connectionString
    = "Server=(localdb)\\... shortened to fit";
var builder = new
    DbContextOptionsBuilder
        <DbContextOnConfiguring>();
builder.UseSqlServer(connectionString);
var options = builder.Options;
using (var context = new
    DbContextOnConfiguring(options)
{
    //… unit test starts here
```

Настраивает опции, которые вы хотите использовать

Предоставляет опции для DbContext через новый конструктор с одним параметром

Теперь можно приступить к модульному тестированию.

17.3 Три способа смоделировать базу данных при тестировании приложений EF Core

Если вы выполняете модульное тестирование приложения и оно включает доступ к базе данных, есть несколько способов ее моделирования. За прошедшие годы я попробовал несколько подходов к моделированию баз данных в модульном тесте, от библиотеки, имитирующей DbContext в EF6, под названием Effort (https://entityframework-effort.net/overview) до использования реальных баз данных. В этой главе рассматриваются некоторые из этих подходов и несколько новых тактик, предлагаемых EF Core.

ПРИМЕЧАНИЕ Подробнее о заглушках и имитациях я расскажу в разделе 17.7.

Ранний выбор способа модульного тестирования базы данных может впоследствии избавить вас от многих проблем, особенно если вы используете EF Core. Когда я начал писать первое издание этой книги, то обнаружил, что подход к модульному тестированию, который я применял вначале, не работает с частями, в большей степени основанными на SQL, поэтому мне пришлось провести рефакторинг некоторых своих ранних модульных тестов, что было немного проблематично.

Но такое было не в первый раз. Позже в некоторых моих проектах я сожалел о своих поспешных решениях относительно модульного тестирования, поскольку эти тесты стали ломаться по мере роста проектов. Хотя некоторая переделка ранних модульных тестов неизбежна, нужно свести к минимуму количество переделок, потому что это замедляет работу. Поэтому я хочу описать различные способы модульного тестирования кода с помощью EF Core, чтобы вы могли принять обоснованное решение относительно того, как писать модульные тесты. На рис. 17.3 показаны три основных способа тестирования кода, содержащего доступ к базе данных.

ОПРЕДЕЛЕНИЕ Термин *рабочая база данных* относится к типу/провайдеру базы данных, используемому вашим приложением в промышленном окружении. Например, если вы запускаете веб-приложение ASP.NET Core с использованием EF Core и это приложение применяет базу данных SQL Server, то рабочая база данных – это SQL Server. В данном случае использование того же типа базы данных, что и в промышленном окружении, означает, что базы данных SQL Server будут применяться и в ваших модульных тестах.

Три способа модульного тестирования кода EF Core с указанием плюсов и минусов

	Используйте тот же тип базы данных, что и у рабочей базы	Используйте базу данных SQLite in-memory	Заглушка базы данных
ПЛЮСЫ:	• Идеально подходит для рабочей базы данных • Обрабатывает SQL	• Быстрый запуск • Есть правильная схема • Запускается пустой	• Обеспечивает полный контроль доступа к данным • Быстрый запуск
МИНУСЫ:	• Нуждается в уникальной базе данных для каждого класса модульного тестирования • Требует времени для создания схемы / пустой базы данных	• Не поддерживает некоторые команды SQL • Не работает как рабочая БД	• Невозможно протестировать некоторый код БД, например связи • Нужно написать больше кода
ЛУЧШЕ ВСЕГО ПОДХОДИТ:	**Когда ваш код содержит чистый SQL**	**Когда в вашем коде используются только команды LINQ**	**Если вы хотите протестировать сложную бизнес-логику**

Рис. 17.3 Есть три основных способа предоставить доступ к базе данных при тестировании кода. У каждого подхода есть свои плюсы и минусы, основные из которых перечислены на рисунке

Нет правильного ответа на вопрос, какой подход лучше, – есть только ряд компромиссов между модульными тестами, работающими точно так же, как ваше рабочее приложение, и временем их написания и запуска. Безопасное решение – использовать базу данных того же типа, что и ваша рабочая база. Но часто при модульном тестировании некоторых приложений я использую комбинацию этих трех подходов.

Прежде чем я опишу три подхода к моделированию базы данных, в разделе 17.4 мы подробнее разберем различия между первыми двумя подходами. Этот раздел дает больше информации, которая поможет вам решить, сможете ли вы протестировать свое приложение с помощью базы данных SQLite in-memory или же необходимо использовать базу данных того же типа, что и ваша рабочая база данных.

17.4 Выбор между базой данных того же типа, что и рабочая, и базой данных SQLite in-memory

В этом разделе приводится информация, необходимая, чтобы решить, какую базу данных использовать: того же типа, что и рабочая, или SQLite in-memory. Следует подумать об использовании базы данных SQLite in-memory, потому что для модульного тестирования проще каждый раз создавать новую базу данных. В результате:

- схема базы данных всегда актуальна;
- база данных пуста, а это хорошая отправная точка для модульного теста;
- параллельное выполнение модульных тестов работает корректно, потому что каждая база данных хранится локально в каждом тесте;
- модульные тесты будут успешно выполняться в тестовой части конвейера DevOps без каких-либо дополнительных настроек;
- ваши модульные тесты быстрее.

Обратная сторона состоит в том, что база данных SQLite не поддерживает и/или не соответствует некоторым SQL-командам из рабочей базы данных, поэтому модульные тесты не сработают или, в некоторых случаях, дадут неверные результаты. Если это вас беспокоит, используйте для модульного тестирования тот же тип базы данных, что и ваша рабочая база (см. раздел 17.5).

Если вы хотите рассмотреть возможность использования SQLite для модульного тестирования, нужно знать, насколько она может отличаться от рабочей базы данных. Простой ответ – «сильно», но чтобы помочь вам понять, что может вызвать проблемы, я подготовил табл. 17.2. В ней перечислены функции, которые могут вызывать проблемы при использовании SQLite для модульного тестирования. В крайнем правом столбце перечислены возможные результаты использования функции:

- *неверный ответ* – функция может сработать, но даст неправильный ответ (что при модульном тестировании худший результат). Вы должны быть внимательны и запустить тест также и с рабочей базой данных или убедиться, что понимаете ограничения, и обходить их;
- *может сломаться* – эта функция может работать правильно в коде модульного теста, но в некоторых случаях может выбросить исключение. Можно протестировать эту функцию с SQLite, но, возможно, придется перейти на базу данных промышленного типа, если модульный тест завершится неудачно;
- *сломается* – функция, скорее всего, приведет к ошибке при настройке базы данных (но может и сработать при базовом SQL). В этом случае использование SQLite in-memory исключено.

Таблица 17.2 Функции SQL, которыми может управлять EF Core, но которые не будут работать с SQLite, поскольку не поддерживаются в SQLite или используют формат, отличный от формата SQL Server, MySQL и т. д.

Функция SQL	См. раздел	Поддержка SQLite?	Вид поломки
Сравнение строк и сопоставления	2.8.3	Работает, но дает разные результаты	Неверный ответ
Разные схемы	7.12.2	Не поддерживается; конфигурация игнорируется	Неверный ответ
Значение столбца SQL по умолчанию	10.3	Константы C# работают; SQL, скорее всего, приведет к ошибке	Может сломаться

Таблица 17.2 (*окончание*)

Функция SQL	См. раздел	Поддержка SQLite?	Вид поломки
Вычисляемые столбцы SQL	10.2	Другой SQL; вероятно, приведет к ошибке	Сломается
Любой чистый SQL-код	11.5	Другой SQL; очень вероятно, что приведет к ошибке	Сломается
SQL-последовательности	10.4	Исключение о неподдерживаемой функции	Сломается

Кроме того, следующие типы C# изначально не поддерживаются SQLite, поэтому могут возвращать неправильное значение:

- Decimal;
- UInt64;
- DateTimeOffset;
- TimeSpan.

EF Core выбросит исключение, например, если вы отсортируете или отфильтруете свойство типа `Decimal` при работе в SQLite. Если вы все еще хотите провести модульное тестирование с помощью SQLite, то можно добавить преобразователь значений для преобразования `Decimal` в `double`, но этот подход может не вернуть точное значение `Decimal`, сохраненное в базе данных.

Поэтому если вы используете какие-либо функции из табл. 17.2, которые не работают, вам определенно не захочется использовать SQLite для модульного тестирования. Но, кроме этого, необходимо подумать и о том, что вы планируете добавить в свое приложение, потому что если вы добавите код, использующий функции из разряда «сломается», вам придется изменить все свои модульные тесты, чтобы использовать базу данных того же типа, что и ваша рабочая база данных, а это может стать настоящей головной болью.

Если вы не используете и вряд ли будете использовать функциональные возможности, приводящие к «поломкам», как показано в табл. 17.2, SQLite может стать хорошим выбором для большинства ваших модульных тестов. Вы можете переключиться на использование тестовой базы данных промышленного типа для выявления функций вида «может сломаться», что я и делаю для приложений EF Core, которые не так широко используют возможности чистого SQL.

> **ПРИМЕЧАНИЕ** Я еще не рассматривал плюсы и минусы третьего варианта, изображенного на рис. 17.3: заглушка базы данных (см. раздел 17.7). Заглушка базы данных – это подход, отличный от использования SQLite или базы данных промышленного типа, поскольку он пытается полностью исключить использование базы данных из модульных тестов. По этой причине заглушка базы данных не проверяет ваш код EF Core. Поэтому я начну с двух подходов, которые включают EF Core: использование базы данных SQLite in-memory и использование базы данных промышленного типа.

17.5 Использование базы данных промышленного типа в модульных тестах

В этом разделе рассматривается использование базы данных промышленного типа, что является лучшим способом модульного тестирования, поскольку ваши базы данных модульного тестирования полностью совместимы с рабочей базой данных. Есть и обратная сторона: такую базу данных сложнее настроить, чем базу SQLite in-memory (см. раздел 17.6), а еще она немного медленнее работает. Чтобы использовать базу данных промышленного типа в модульных тестах, необходимо решить четыре проблемы:

- настроить строку подключения к базе данных, которая будет использоваться в модульном тесте;
- создать базу данных для каждого тестового класса для параллельного запуска тестов в xUnit;
- убедиться, что схема базы данных актуальна, а база данных пуста;
- имитировать настройку базы данных, которую обеспечивает миграция EF Core.

Интересно, что подход с базой данных SQLite in-memory решает первые три проблемы из списка, а последний пункт, связанный с SQL, встроенным в ваши миграции, – это то, с чем SQLite не может справиться, потому что код SQL, скорее всего, будет другим. Список из четырех проблем, с которыми вы должны справиться, чтобы запустить модульный тест, – верный признак дополнительных усилий, связанных с поиском лучшего способа модульного тестирования кода для работы с базами данных. Вам поможет моя библиотека EfCore.TestSupport, предоставляющая методы расширения, которые помогут настроить параметры базы данных, решить проблему «база данных для каждого тестового класса» и убедиться, что схема базы данных актуальна и база данных пуста.

> **ПРИМЕЧАНИЕ** В следующих примерах используется база данных SQL Server, но эти подходы одинаково хорошо работают с типами баз данных, отличными от Cosmos DB, которой посвящен отдельный раздел (17.8).

17.5.1 Настройка строки подключения к базе данных, которая будет использоваться для модульного теста

Для доступа к любой базе данных нужна строка подключения (см. раздел 5.4.1). Можно было бы определить строку подключения как константу и использовать ее, но, как вы увидите, этот подход не так гибок, как хотелось бы. Поэтому в данном разделе мы сымитируем то, что делает ASP.NET Core, добавив в наш тестовый проект простой файл appsettings.json, содержащий строку подключения. Затем мы

воспользуемся пакетами конфигурации .NET для доступа к строке подключения в нашем приложении. Файл appsettings.json выглядит примерно так:

```
{
  "ConnectionStrings": {
    "UnitTestConnection": "Server=(localdb)\\mssqllocaldb;Database=... etc"
  }
}
```

ПРЕДУПРЕЖДЕНИЕ Не нужно помещать строку подключения, содержащую закрытые ключи, пароли и т. д., в файл appsetting.json, так как эти элементы могут подвергнуться утечке, если вы храните свой код в системе управления версиями. В .NET есть *пользовательские секреты*, встроенные в ASP.NET Core (http://mng.bz/rmYg). Можно использовать их в своих модульных тестах с помощью метода `AddUserSecrets`.

В листинге 17.7 показан метод `GetConfiguration` из моей библиотеки EfCore.TestSupport. Этот метод загружает файл appsettings.json из каталога верхнего уровня вызывающей сборки, т. е. сборки, в которой вы запускаете ваши модульные тесты.

Листинг 17.7 Метод `GetConfiguration`, разрешающий доступ к файлу appsettings.json

Возвращает IConfigurationRoot, из которого можно использовать такие методы, как GetConnectionString («ConnectionName»), для получения данных из конфигурации

В библиотеке TestSupport метод возвращает абсолютный путь к каталогу верхнего уровня вызывающей сборки (сборка, в которой запускаются тесты)

```
public static IConfigurationRoot GetConfiguration()
{
    var callingProjectPath =
        TestData.GetCallingAssemblyTopLevelDir();
    var builder = new ConfigurationBuilder()
        .SetBasePath(callingProjectPath)
        .AddJsonFile("appsettings.json", optional: true);
    return builder.Build();
}
```

Вызывает метод Build, который возвращает тип IConfigurationRoot

Использует ConfigurationBuilder от ASP.NET Core для чтения файла appsettings.json. Это необязательно, поэтому ошибка не возникнет, если файла конфигурации не существует

Можно использовать метод `GetConfigration` для доступа к строке подключения, а затем использовать следующий код для создания DbContext:

```
var config = AppSettings.GetConfiguration();
config.GetConnectionString("UnitTestConnection");
var builder = new DbContextOptionsBuilder<EfCoreContext>();
```

```
builder.UseSqlServer(connectionString);
using var context = new EfCoreContext(builder.Options);
// … Остальная часть модульного теста не указана;
```

Такой код решает проблему получения строки подключения, но проблема наличия разных баз данных для каждого тестового класса остается, потому что по умолчанию xUnit параллельно запускает модульные тесты. Эта тема рассматривается в разделе 17.5.2.

17.5.2 Создание базы данных для каждого тестового класса для параллельного запуска тестов в xUnit

Поскольку xUnit может запускать каждый класс модульных тестов параллельно, использовать общую базу данных для всех тестов не получится. Хорошие модульные тесты нуждаются в известной отправной точке и должны вернуть известный результат, что исключает использование одной базы данных, поскольку разные тесты могут одновременно вносить изменения в базу данных.

> **ПРИМЕЧАНИЕ** Можно запускать xUnit последовательно (см. раздел *Changing Default Behavior* (Изменение поведения по умолчанию) на странице https://xunit.net/docs/running-tests-in-parallel), но я не рекомендую этого делать, потому что это замедлит выполнение тестов.

Одно из распространенных решений – создавать разноименные базы данных для каждого класса модульного тестирования или, возможно, каждого метода модульного тестирования. Библиотека EfCore.TestSupport содержит методы, которые создают SQL Server `DbContextOptions<T>`, где имя базы данных уникально для тестового класса или метода. На рис. 17.4 показаны два метода. Первый метод создает базу данных с именем, уникальным для этого класса, а второй создает базу данных с именем, уникальным для класса и метода.

В результате использования любого из этих классов каждый тестовый класс или метод имеет собственную базу данных с уникальным именем. Поэтому когда модульные тесты выполняются параллельно, у каждого тестового класса есть собственная база данных для тестирования.

> **СОВЕТ** xUnit запускает каждый тестовый класс параллельно; но внутри класса запускает каждый тест последовательно. По этой причине я обычно использую базу данных, уникальную для класса. Я редко использую базу данных, уникальную для каждого класса и метода, но если она мне понадобится, она уже будет под рукой.

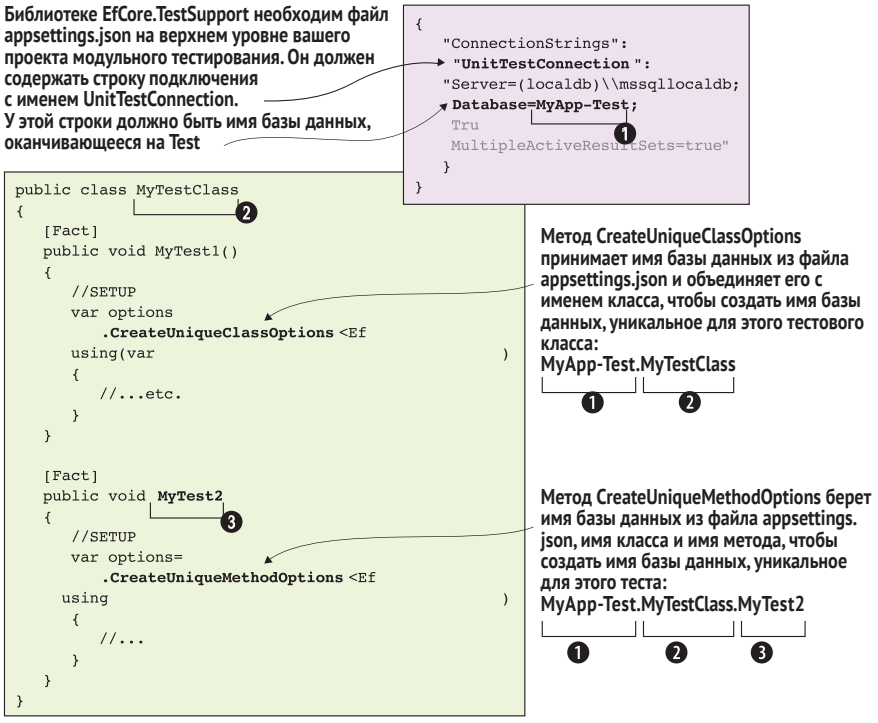


Рис. 17.4 Два метода, которые настраивают параметры базы данных для базы данных SQL Server, но изменяют имя базы, чтобы оно было уникальным для класса или уникальным для класса и метода. Когда вы запускаете несколько модульных тестов, у них будут свои базы данных, поэтому они не будут мешать друг другу

В следующем листинге показан код внутри метода расширения `CreateUniqueClassOptions`. Этот код инкапсулирует все настройки параметров DbContext, чтобы избавить вас от необходимости включать их в каждый модульный тест.

Листинг 17.8 Метод расширения `CreateUniqueClassOptions` и вспомогательный метод

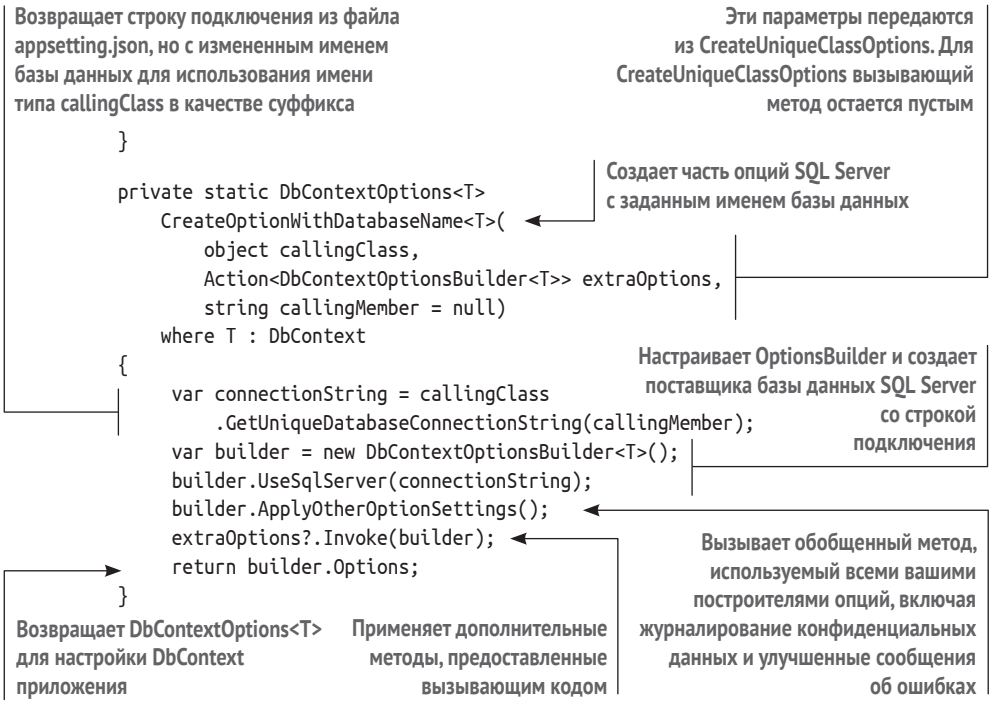
Ожидается, что предоставленный экземпляр объекта будет this – классом, в котором выполняется модульный тест

Возвращает параметры для базы данных SQL Server с именем, начинающимся с имени базы данных в исходной строке подключения из файла appsettings.json, но с именем класса экземпляра, указанным в первом параметре

```
public static DbContextOptions<T>
    CreateUniqueClassOptions<T>(
        this object callingClass,
        Action<DbContextOptionsBuilder<T>> builder = null)
    where T : DbContext
{
    return CreateOptionWithDatabaseName<T>
        (callingClass, builder);
```

Этот параметр позволяет добавлять дополнительные методы настройки при создании опций

Вызывает закрытый метод, совместно используемый этим методом и параметрами CreateUniqueMethodOptions



У параллельного запуска в xUnit есть и другие ограничения. Использование статических переменных для передачи информации вызывает проблемы, например разные тесты могут параллельно устанавливать для статической переменной разные значения. В наши дни мы редко используем статические переменные, потому что внедрение зависимостей заполняет этот пробел. Но если вы используете их в своем коде, то должны отключить параллельный запуск в xUnit, чтобы тесты выполнялись последовательно.

17.5.3 Убеждаемся, что схема базы данных актуальна, а база данных пуста

В разделе 17.5.2 показано, как создавать уникальные базы данных для тестов, но все еще есть проблема: нужно убедиться, что база данных пуста и соответствует актуальной схеме при повторном запуске теста. Когда я говорю, что «схема базы данных актуальна», я имею в виду, что схема базы данных соответствует модели базы данных, которую EF Core создает путем сканирования ваших классов сущностей и любого кода конфигурации, который вы применили к DbContext приложения.

В отличие от приложения, которое будет использовать некую форму миграции для обновления классов сущностей или конфигурации EF Core, модульные тесты будут использовать методы EF Core `EnsureCreated` и `EnsureCreatedAsync`, чтобы убедиться, что база данных

существует. Эти методы создают базу данных с актуальной схемой, используя текущие классы сущностей и конфигурацию DbContext приложения, но только если уже не создана база данных с таким же именем. После первого запуска теста схема базы данных фиксируется, поэтому ее схема будет устаревшей, если вы измените конфигурацию EF Core или какой-либо класс сущности. Следовательно, нужен способ убедиться, что схема базы данных актуальна, и в то же время предоставить пустую базу в качестве отправной точки для модульного теста.

Начнем с надежного, но медленного метода. В листинге 17.9 показан рекомендуемый Microsoft способ создания пустой базы данных с правильной схемой без использования миграций. Эти два метода EF Core удаляют и создают базу данных; метод создания базы данных создает схему базы данных, используя текущую конфигурацию EF Core и классы сущностей. В следующем листинге сначала показан вызов метода `EnsureDeleted` для удаления базы данных, а затем вызов метода EnsureCreated для создания правильной схемы базы данных.

Листинг 17.9 Надежный способ создать актуальную и пустую базу данных

```
[Fact]
public void TestExampleSqlDatabaseOk()
{
    //НАСТРОЙКА
    var options = this
        .CreateUniqueClassOptions<EfCoreContext>();
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureDeleted();
        context.Database.EnsureCreated();
        //… Остальная часть теста удалена;
```

Удаляет текущую базу данных (если она есть)

Создает новую базу данных, используя конфигурацию внутри DbContext приложения

Поскольку в листинге 17.9 используется метод `CreateUniqueClassOptions` из библиотеки EfCore.TestSupport, каждый модульный тест в этом классе использует одну и ту же базу данных, но каждый метод модульного теста удаляет и создает базу данных заново на этапе настройки теста.

Раньше этот подход был медленным (~ 10 секунд) для базы данных SQL Server, но поскольку в .NET 5 появилась новая библиотека SqlClient, теперь все стало намного быстрее (~ 1.5 секунды). Это сильно влияет на время выполнения модульного теста с версией `EnsureDeleted/EnsureCreated`.

ПРИМЕЧАНИЕ Сколько времени это займет, зависит от базы данных. Когда я писал первое издание данной книги, удаление и создание базы данных SQL Server раньше занимало

около 10 секунд, но с базой данных MySQL занимало всего секунду. Нужно протестировать ваш тип базы данных, чтобы узнать, сколько времени требуется, чтобы удалить и создать базу данных.

Еще один подход, предложенный Артуром Викерсом из команды EF Core, – это метод, который команда использует в собственных модульных тестах: `EnsureClean`. Этот метод удаляет текущую схему базы данных путем удаления всех индексов SQL, ограничений, таблиц, последовательностей, пользовательских функций и т. д. из базы данных. Затем по умолчанию вызывается метод `EnsureCreated`, чтобы создать базу данных, у которой правильная схема и которая не содержит данных.

Метод `EnsureClean` находится глубоко в модульных тестах EF Core, но я извлек его код и создал другие части, необходимые для того, чтобы сделать его полезным; он доступен в библиотеке EfCore.TestSupport версии 5. В следующем листинге показано, как использовать его в модульных тестах.

Листинг 17.10 Использование метода `EnsureClean` для обновления схемы базы данных

```
[Fact]
public void TestExampleSqlServerEnsureClean()
{
    //НАСТРОЙКА
    var options = this.
        CreateUniqueClassOptions<BookDbContext>();

    using var context = new BookDbContext(options);

    context.Database.EnsureClean();

    //… Остальная часть теста удалена;
}
```

Удаляет данные и схему в базе данных, а затем вызывает метод EnsureCreated для настройки правильной схемы

Подход с `EnsureClean` быстрее, может быть, вдвое быстрее, чем версия с `EnsureDeleted/EnsureCreated`, что может сильно повлиять на скорость выполнения модульных тестов. Кроме того, такой подход лучше, когда сервер базы данных не позволяет вам удалять или создавать новые базы данных, но позволяет выполнять операции чтения и записи в базу данных, например когда тестовые базы данных находятся на сервере SQL, на котором у вас нет прав администратора.

ПРИМЕЧАНИЕ На данный момент метод `EnsuredClean` работает только для SQL Server, но его можно улучшить для работы с другими типами баз данных. Однако если для типа базы данных уже реализовано быстрое выполнение `EnsureDeleted/EnsureCreated`, расширять его не стоит.

Последний подход к получению базы данных для использования в модульном тесте необычен, но может быть полезен в некоторых ситуациях. Он применяет изменения к базе данных только в рамках транзакции. Этот подход работает, потому что при отмене транзакции, если вы не вызвали метод `transaction.Commit`, он откатывает все изменения, внесенные в базу данных, пока транзакция активна. В результате каждый раз каждый модульный тест начинается с одних и тех же данных.

Это полезный подход, если у вас есть эталонная база данных, возможно скопированная с рабочей базы данных (разумеется, с анонимизированными личными данными), которую вы хотите протестировать, но не хотите, чтобы изменения сохранялись в базе. Я использовал его для клиента, у которого был пример базы данных (размером 1 ТБ и хранящейся в Azure). С помощью версии с транзакцией мне удалось запустить часть клиентского кода, дабы понять, что изменилось в базе данных без изменения ее содержимого.

Чтобы использовать версию с транзакцией, нужно создать транзакцию сразу после создания DbContext приложения и сохранить ее в переменной, которая будет удалена в конце модульного теста. В следующем листинге я добился этого эффекта с помощью ключевых слов `using var`.

Листинг 17.11 Использование транзакции для отката любых изменений базы данных, сделанных в тесте

```
[Fact]
public void TestUsingTransactionToRollBackChanges()
{
    //НАСТРОЙКА
    var builder = new
        DbContextOptionsBuilder<BookDbContext>();
    builder.UseSqlServer(_connectionString);
    using var context =
        new BookDbContext(builder.Options);

    using var transaction =
        context.Database.BeginTransaction();

    //ДЕЙСТВИЕ
    var newBooks = BookTestData
        .CreateDummyBooks(10);
    context.AddRange(newBooks);
    context.SaveChanges();

    //ПРОВЕРКА
    context.Books.Count().ShouldEqual(4+10);
}
```

Скорее всего, вы будете связываться с базой данных, используя строку подключения

Транзакция хранится в переменной, объявленной как using var, а это означает, что она будет удалена, когда текущий блок закончится

Запускаем тест...

... и проверяем, сработал ли он

Когда метод модульного тестирования завершит работу, транзакция будет удалена и откатит изменения, сделанные в модульном тесте. В данном случае четыре книги уже были в базе

17.5.4 Имитация настройки базы данных, которую обеспечит миграция EF Core

Одна проблема, с которой я столкнулся при модульном тестировании, возникла, когда в моей базе данных были дополнительные SQL-команды, которые были добавлены в обход EF Core. Например, как использовать SQL-код в базе данных модульного теста, если вы используете пользовательские функции в своем коде? Есть три решения:

- для простого SQL-кода, такого как пользовательские функции, можно выполнить файл сценария после метода `EnsureCreated`;
- если вы добавили свой SQL в файлы миграции EF Core (см. раздел 9.5.2), то нужно вызывать context.Database.Migrate вместо ....`EnsureCreated`;
- если вы используете миграции на базе сценариев (см. раздел 11.4), то вместо вызова метода `EnsureCreated` следует выполнить сценарии для создания базы данных.

Последние два пункта уже подробно описаны, но для первого потребуется дополнительный код. Я создал метод `ExecuteScriptFileInTransaction` в моей библиотеке EfCore.TestSupport. Этот метод выполняет SQL-код файла сценария SQL в базе данных, к которой подключен DbContext приложения. Формат сценария – Microsoft SQL Server Management Studio: набор SQL-команд, каждая из которых заканчивается строкой, содержащей команду `GO`. В следующем листинге показан файл сценария SQL, добавляющий пользовательскую функцию в базу данных.

Листинг 17.12 Пример файла сценария SQL с командой `GO` в конце каждой SQL-команды

Удаляет существующую версию пользовательской функции, которую вы хотите добавить. Если этого не сделать, функция создания не сработает

```
IF OBJECT_ID('dbo.AuthorsStringUdf') IS NOT NULL
    DROP FUNCTION dbo.AuthorsStringUdf
GO

CREATE FUNCTION AuthorsStringUdf (@bookId int)
RETURNS NVARCHAR(4000)
-- … Команды SQL удалены, чтобы сделать этот пример короче;
RETURN @Names
END
GO
```

ExecuteScriptFileInTransaction ищет строку, начинающуюся с GO, чтобы разделить каждую команду SQL для отправки в базу данных

Добавляет в базу данных пользовательскую функцию

Метод расширения `ExecuteScriptFileInTransaction` может применить сценарий SQL к базе данных, используя формат из листинга 17.12. В листинге 17.13 показан типичный способ применения этого сценария к базе данных модульного тестирования.

ПРИМЕЧАНИЕ `TestData.GetFilePath` из следующего листинга – еще один метод из библиотеки EfCore.TestSupport; он позволяет получить доступ к файлам из каталога верхнего уровня TestData в проекте Test.

Листинг 17.13 Пример применения сценария SQL к базе данных модульного теста

```
[Fact]
public void TestApplyScriptExampleOk()
{
    var options = this
        .CreateUniqueClassOptions<EfCoreContext>();
    var filepath = TestData.GetFilePath(          | Получает путь к файлу сценария SQL
        "AddUserDefinedFunctions.sql");           | с помощью метода TestData.GetFilePath
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureDeleted();
        context.Database.EnsureCreated();
        context                                   | Применяет сценарий к базе
            .ExecuteScriptFileInTransaction(      | данных с помощью метода
            filepath);                            | ExecuteScriptFileInTransaction
   //… Остальная часть модульного теста не показана;
    }
}
```

17.6 Использование базы данных SQLite in-memory для модульного тестирования

В SQLite есть полезная опция для создания базы данных в памяти (*in-memory*). Эта опция позволяет модульному тесту создать новую изолированную базу данных в памяти. Такой подход решает все проблемы, связанные с запуском параллельных тестов, наличием актуальной схемы и обеспечением того, чтобы база данных была пустой, кроме того, он работает быстро. Но см. раздел 17.4, где рассказывается о потенциальных проблемах.

Чтобы создать такую базу данных, необходимо задать для `DataSource` значение `":memory:"`, как показано далее. Код в листинге 17.14 взят из метода `SqliteInMemory.CreateOptions` из моей библиотеки EfCore.TestSupport.

ПРИМЕЧАНИЕ Метод `CreateOptions` из листинга 17.14 возвращает класс `DbContextOptionsDisposable<T>`. Этот класс реализует тип `DbContextOptionsBuilder<T>`, необходимый для создания экземпляра DbContext приложения, и интерфейс `IDisposable`,

который используется для удаления подключения к SQLite при удалении DbContext приложения. Я расскажу об этом ближе к концу раздела.

Листинг 17.14 Создание параметров DbContextOptions<T> базы данных SQLite в памяти

```
public static DbContextOptionsDisposable<T> CreateOptions<T>
    (Action<DbContextOptionsBuilder<T>> builder = null)
    where T : DbContext
{
    return new DbContextOptionsDisposable<T>
       (SetupConnectionAndBuilderOptions<T>(builder)
           .Options);
}

private static DbContextOptionsBuilder<T>
    SetupConnectionAndBuilderOptions<T>
    (Action<DbContextOptionsBuilder<T>> applyExtraOption)
    where T : DbContext
{
    var connectionStringBuilder =
        new SqliteConnectionStringBuilder
            { DataSource = ":memory:" };
    var connectionString = connectionStringBuilder.ToString();
    var connection = new SqliteConnection(connectionString);
    connection.Open();

    // Создаем контекст в памяти;
    var builder = new DbContextOptionsBuilder<T>();
    builder.UseSqlite(connection);
    builder.ApplyOtherOptionSettings();
    applyExtraOption?.Invoke(builder);

    return builder;
}
```

Этот параметр позволяет добавлять дополнительные методы настройки при создании опций

Класс, содержащий параметры SQLite в памяти, который также является высвобождаемым (disposable)

Этот метод создает параметры SQLite для работы в памяти

Получает DbContextOptions<T> и возвращает высвобождаемую (disposable) версию

Содержит любые дополнительные методы, заданные внешним кодом

Создает строку подключения из SQLiteConnection-StringBuilder

Создает строку подключения к SQLite с DataSource со значением «:memory:»

Формирует соединение с SQLite с помощью строки подключения

Вы должны открыть подключение к SQLite. В противном случае база данных в памяти работать не будет

Создает DbContextOptions<T> с поставщиком базы данных SQLite и открытым соединением

Добавляет все дополнительные опции, добавленные пользователем

Возвращает DbContextOptions<T>, который будет использоваться при создании DbContext приложения

Вызывает обобщенный метод, используемый всеми построителями опций, что позволяет вести журналирование конфиденциальных данных и улучшать сообщения об ошибках

После этого можно использовать метод `SQLiteInMemory.CreateOptions` в одном из модульных тестов, как показано в следующем листинге. Обратите внимание, что в данном случае необходимо вызвать только метод `EnsureCreated`, поскольку в настоящее время базы данных еще нет.

Листинг 17.15 Использование базы данных SQLite in-memory в модульном тесте xUnit

```
[Fact]
public void TestSQLiteOk()
{
    //НАСТРОЙКА
    var options = SQLiteInMemory
        .CreateOptions<EfCoreContext>();

    using var context = new BookDbContext(options);

    context.Database.EnsureCreated();

    //ДЕЙСТВИЕ
    context.SeedDatabaseFourBooks();

    //ПРОВЕРКА
    context.Books.Count().ShouldEqual(4);
}
```

SQLiteInMemory.CreateOptions предоставляет опции для базы данных в памяти. Эти опции тоже IDisposable

Использует эту опцию для создания DbContext приложения

Вызываем context.Database.EnsureCreated для создания базы данных

Запускает написанный вами метод тестирования, который добавляет в базу данных четыре тестовые книги

Проверяет, что SeedDatabaseFourBooks сработал, и добавляет четыре книги в базу данных

В конце модульного теста context удаляется, потому что вы использовали оператор `using var` для хранения экземпляра DbContext. Удаление контекста, в свою очередь, удаляет переменную options, которая удаляет базу данных, избавляясь от подключения `SqliteConnection`. Удаление этого подключения выполняется в соответствии с рекомендуемыми практиками в документации EF Core: http://mng.bz/VG7X.

ПРИМЕЧАНИЕ Если вы используете несколько экземпляров DbContext, нужно отложить удаление подключения `SqliteConnection` с помощью методов `options.StopNextDispose` или `options.TurnOffDispose` (см. раздел 17.10.2, где приводится один способ).

Что насчет провайдера базы данных в памяти EF Core для модульного тестирования?

В EF Core есть провайдер базы данных в памяти, который команда использует при тестировании, но в документации указано, что этот провайдер «не подходит для тестирования приложений, использующих EF Core» (http://mng.bz/xGO8). Поэтому команда была удивлена, получив отзывы, где говорилось, что многие люди используют провайдера базы данных в памяти для модульного тестирования.

Когда я писал первое издание этой книги, то использовал провайдера базы данных в памяти и быстро наткнулся на ограничения. Во-первых, он работает не как настоящая реляционная база данных; следовательно, он не находит некоторые проблемы. Когда я обнаружил, что у SQLite есть режим «в памяти», я переключился на эту базу данных. Не идеальный вариант, но это *гораздо* лучше, чем провайдер базы данных в памяти EF Core.

17.7 Создание заглушки или имитации базы данных EF Core

После реальной базы данных рассмотрим третий подход, изображенный на рис. 17.3: заглушка или имитация базы данных. Вот определения двух этих подходов:

- *создание заглушки* для базы данных означает создание кода, заменяющего текущую базу данных. Заглушка хорошо работает при использовании паттерна «Репозиторий» (см. раздел 13.5.1);
- для *имитации* обычно требуется специальная библиотека, например Moq (https://github.com/moq/moq4), используемая для управления классом, который вы имитируете. Для EF Core эта задача в принципе невозможна; наиболее подходящая библиотека для имитации в EF Core – это провайдер базы данных в памяти EF Core.

ПРИМЕЧАНИЕ В этой статье содержится дополнительная информация о заглушках и имитациях: http://mng.bz/A1Wp.

Как мы выяснили выше, имитация не сработает, поэтому далее я покажу пример, который использую со сложной бизнес-логикой, описанной в разделе 4.2. Здесь я использую паттерн «Репозиторий». Поскольку бизнес-логика может быть сложной, часто со сложными правилами валидации, я считаю, что заглушка – это полезный подход к замене доступа к базе данных. Заглушка обеспечивает гораздо больший контроль над доступом к базе данных, и можно с легкостью моделировать различные ошибочные состояния, но на написание модульных тестов уходит больше времени.

В качестве примера этого подхода мы создадим заглушку для базы данных при тестировании бизнес-логики, обрабатывающей заказы на книги. Метод этой бизнес-логики использует паттерн «Репозиторий» для отделения кода доступа к базе данных от бизнес-логики, упрощая код бизнес-логики; кроме того, это помогает при модульном тестировании, потому что я могу заменить код доступа к базе данных тестовым классом, который может заменить базу данных заглушкой, соответствующей интерфейсу репозитория. На мой взгляд, заглушка дает гораздо лучший контроль над исходящими и входящими данными метода, который я тестирую.

Следующий пример взят из моих модульных тестов из репозитория книги на GitHub; здесь нам нужно протестировать метод `PlaceOrderAction`, разработанный в главе 4. Конструктор класса `PlaceOrderAction` принимает один параметр типа `IPlaceOrderDbAccess`. Обычно это класс `PlaceOrderDbAccess`, обрабатывающий доступ к базе данных. Но для тестирования мы заменяем класс `PlaceOrderDbAccess` тестовым классом-заглушкой, реализующим тот же интерфейс `IPlaceOrderDbAccess`. Этот класс-заглушка позволяет протестировать, что класс `Pla-`

ceOrderAction может читать из базы данных и перехватывать то, что он пытается записать в нее. В следующем листинге показан модульный тест, использующий имитацию, которая перехватывает заказ, производимый методом PlaceOrderAction, чтобы можно было проверить, правильно ли задан идентификатор пользователя.

Листинг 17.16 Модульный тест, предоставляющий экземпляр заглушки для BizLogic

```
[Fact]
public void ExampleOfStubbingOk()
{
    //НАСТРОЙКА
    var lineItems = new List<OrderLineItem>
    {
        new OrderLineItem {BookId = 1, NumBooks = 4}
    };
    var userId = Guid.NewGuid();
    var input = new PlaceOrderInDto(true, userId,
        lineItems.ToImmutableList());

    var stubDbA = new StubPlaceOrderDbAccess();
    var service = new PlaceOrderAction(stubDbA);

    //ДЕЙСТВИЕ
    service.Action(input);

    //ПРОВЕРКА
    service.Errors.Any().ShouldEqual(false);
    mockDbA.AddedOrder.CustomerId
        .ShouldEqual(userId);
}
```

Создает экземпляр фиктивного кода доступа к базе данных. У этого экземпляра множество параметров, но в данном случае используются настройки по умолчанию

Создает входные данные для метода PlaceOrderAction

Создает экземпляр PlaceOrderAction, предоставляя имитацию кода доступа к базе данных

Запускает метод Action для PlaceOrderAction, который принимает входные данные и выводит заказ

Проверяет, что размещение заказа успешно завершено

Ваш фиктивный код доступа к базе данных перехватил заказ, который метод PlaceOrderAction «записал» в базу данных, поэтому вы можете проверить, правильно ли он сформирован

У класса-заглушки StubPlaceOrderDbAccess нет доступа к базе данных, но у него есть свойства или методы, которые можно использовать для управления каждой частью операции чтения данных из базы. Кроме того, этот класс фиксирует все, что метод PlaceOrderAction пытается записать в базу данных для последующей проверки. В листинге 17.17 показан класс-заглушка StubPlaceOrderDbAccess. Обратите внимание, что я создал статический метод CreateDummyBooks, чтобы сгенерировать набор книг для использования в этом тесте (см. раздел 17.9).

Листинг 17.17 Код доступа к базе данных, используемой для модульного тестирования

```
public class StubPlaceOrderDbAccess
    : IPlaceOrderDbAccess
{
    public ImmutableList<Book> DummyBooks
        { get; private set; }

    public Order AddedOrder { get; private set; }

    public StubPlaceOrderDbAccess(
        bool createLastInFuture = false,
        int? promoPriceFirstBook = null)
    {
        var numBooks = createLastInFuture
            ? DateTime.UtcNow.Year -
                EfTestData.DummyBookStartDate.Year + 2
            : 10;
        var books = EfTestData.CreateDummyBooks
            (numBooks, createLastInFuture);
        if (promotionPriceForFirstBook != null)
            books.First().Promotion = new PriceOffer
            {
                NewPrice = (int) promoPriceFirstBook,
                PromotionalText = "Unit Test"
            };
        DummyBooks = books.ToImmutableList();
    }

    public IDictionary<int, Book>
        FindBooksByIdsWithPriceOffers
        (IEnumerable<int> bookIds)
    {
        return DummyBooks.AsQueryable()
            .Where(x => bookIds.Contains(x.BookId))
            .ToDictionary(key => key.BookId);
    }

    public void Add(Order newOrder)
    {
        AddedOrder = newOrder;
    }
}
```

Будет содержать заказ, созданный методом PlaceOrderAction

MockPlaceOrderDbAccess реализует IPlaceOrderDbAccess, что позволяет ему заменить обычный класс PlaceOrderDbAccess

Содержит фиктивные книги, которые используются для имитации, их можно использовать для сравнения результата с фиктивной базой данных

В этом случае вы настраиваете имитацию через конструктор

Позволяет проверить, что еще не опубликованная книга не будет принята в заказ

Позволяет добавить PriceOffer к первой книге, чтобы можно было проверить, что в заказе указана правильная цена

Определяет, как создать достаточно книг таким образом, чтобы последняя была не опубликована

Создает метод для создания фиктивных книг для теста

При необходимости добавляет PriceOffer к первой книге

Вызывается, чтобы получить книги, выбранные при вводе данных; использует DummyBooks, сгенерированные в конструкторе

Код аналогичен оригиналу, но в этом случае чтение осуществляется из DummyBooks, а не из базы данных

Вызывается методом PlaceOrderAction для записи заказа в базу данных. В этом случае вы перехватываете заказ, чтобы модульный тест мог его проверить

Как я сказал ранее, код заглушки длинный и немного сложный для написания, но если скопировать оригинальный класс `PlaceOrderDbAccess`, а затем отредактировать его, работа не такая уж сложная.

17.8 Модульное тестирование базы данных Cosmos DB

Модульное тестирование базы данных Cosmos DB не соответствует ни одному из трех подходов, описанных в разделе 17.4, но ближе всего к имитации базы данных, потому что Microsoft разработала приложение под названием Azure Cosmos DB Emulator, которое можно запускать на компьютере разработчика и проводить тестирование. В документации Microsoft на странице http://mng.bz/RK8j сказано:

> *Azure Cosmos DB Emulator обеспечивает точную эмуляцию сервиса Azure Cosmos DB. Он поддерживает функциональные возможности, эквивалентные Azure Cosmos DB, включая создание данных, запрос данных, подготовку и масштабирование контейнеров и выполнение хранимых процедур и триггеров.*

Необходимо скачать эмулятор Azure Cosmos DB Emulator со страницы http://mng.bz/4MOj и запустить его локально. Когда вы запускаете эмулятор, он предоставляет URL-адрес, который приведет вас к странице быстрого запуска эмулятора, где содержится подробная информация о доступе к этому сервису Cosmos DB. На сайте эмулятора также есть полезный проводник (см. рис. 17.5), обеспечивающий полный доступ и настройку баз данных, контейнеров и элементов внутри контейнера.

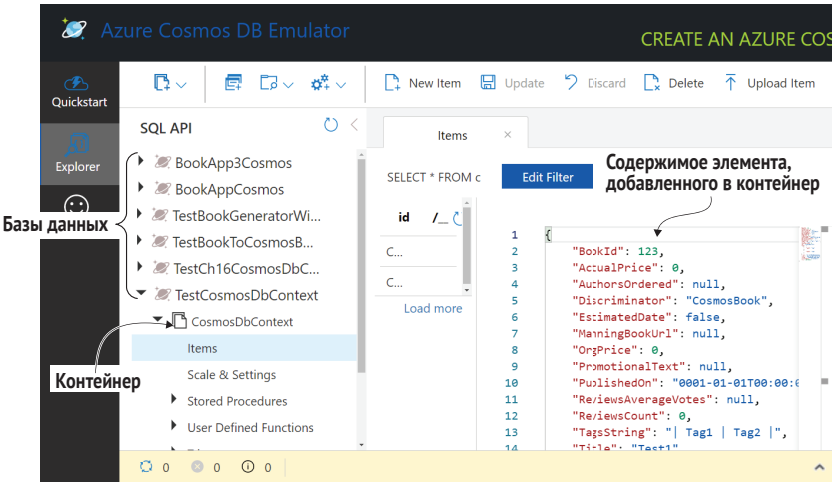


Рис. 17.5 Когда вы запускаете Azure Cosmos DB Emulator, он дает вам URL-адрес для доступа к информации о параметрах эмулятора Cosmos DB на странице быстрого запуска, а также доступа к эмулированным базам данных и контейнерам Cosmos DB через страницу Explorer, как показано на этом рисунке. Страница проводника обеспечивает полный доступ и настройку баз данных, контейнеров и элементов в контейнере

ПРИМЕЧАНИЕ Azure Cosmos DB Emulator доступен только для Windows.

В разделе 16.5 вы узнали, что для доступа к службе Cosmos DB необходима строка подключения. Она указана на странице быстрого запуска Emulator. Все показано в следующем листинге.

Листинг 17.18 Настройка EF Core для доступа к базе данных Cosmos DB

```
public async Task AccessCosmosEmulatorViaConnectionString()
{
    //НАСТРОЙКА
    var connectionString =
        "AccountEndpoint=https://localhost… rest left out"
    var builder = new
          DbContextOptionsBuilder<CosmosDbContext>()
        .UseCosmos(
            connectionString,
            "MyCosmosDatabase");
    using var context = new CosmosDbContext(builder.Options);

    //… Остальная часть модульного теста не показана;
}
```

Строка подключения, взятая со страницы быстрого запуска сайта эмулятора

Создает опции для CosmosDbContext

Сначала предоставляется строка подключения

Имя базы данных

Метод UseCosmos находится в пакете NuGet, Microsoft.EntityFrameworkCore.Cosmos

Создает экземпляр DbContext приложения

Этот подход прекрасно работает, но поскольку строка подключения везде одна и та же, где бы вы ни запустили эмулятор, вы можете создать метод для автоматической настройки параметров. Я добавил методы этого типа в пакеты NuGet EfCore.TestSupport версии 5. Они используют тот же подход, что и методы SQL Server из EfCore.TestSupport (см. рис. 17.4), где имя класса (и, возможно, имя метода) используется для формирования имени базы данных.

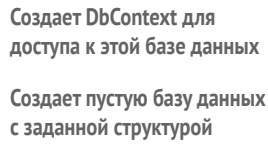
В следующем листинге показано использование метода `CreateUniqueClassCosmosDbEmulator` из библиотеки EfCore.TestSupport, чтобы настроить параметры для DbContext с именем `CosmosDbContext`. Этот код создает базу данных Cosmos с тем же именем, что и у типа класса модульного теста, что делает базу данных уникальной в рамках проекта.

Листинг 17.19 Модульное тестирование кода Cosmos DB с помощью эмулятора Cosmos DB

```
[Fact]
public async Task TestAccessCosmosEmulator()
{
    //НАСТРОЙКА
    var options = this.
       CreateUniqueClassCosmosDbEmulator
        <CosmosDbContext>();
```

Этот метод настраивает параметры базы данных Cosmos DB с именем базы данных, взятым из имени класса

```
    using var context = new CosmosDbContext(options);

    await context.Database.EnsureDeletedAsync();
    await context.Database.EnsureCreatedAsync();
     //… Остальная часть модульного теста не указана;
}
```



Как я уже говорил в разделе 16.6.6, метод `EnsureCreatedAsync` – это рекомендуемый способ создать пустую базу данных Cosmos DB. Следовательно, использование метода `EnsureDeletedAsync`, а затем `EnsureCreatedAsync` – это правильный способ удаления и повторного создания базы данных Cosmos DB. К счастью, это быстро.

17.9 Заполнение базы данных тестовыми данными для правильного тестирования кода

Часто для запуска модульного теста требуются определенные данные в базе данных. Например, чтобы протестировать код, обрабатывающий заказы на книги, перед запуском теста потребуются сущности `Book`. В таких случаях нужно добавить код на этапе настройки модульного теста, чтобы заранее добавить книги перед проверкой кода заказа.

По моему опыту, настройка базы данных с данными для тестирования некоторых функций в реальном приложении может быть довольно сложной. Фактически настроить базу данных с использованием правильного типа может быть намного сложнее, чем запустить тест и проверить его результаты. Вот несколько советов по заполнению базы данных для модульных тестов:

- вначале можно написать код настройки в модульном тесте, но как только вы заметите, что копируете этот код, выносите его в метод;
- я создал два типа вспомогательных методов в моем тестовом проекте для настройки тестовых данных и дал им подходящие имена, чтобы можно было быстро определить, что они делают. Вот эти два типа:
  - методы, возвращающие тестовые данные, с такими именами, как `CreateFourBooks()` и `CreateDummyBooks(int numBooks = 10)`. Я использую эти методы, когда хочу протестировать добавление этих типов в базу данных;
  - методы, записывающие тестовые данные в базу данных, с такими именами, как `SeedDatabaseFourBooks()` и `AddDummyBooksToDb()`. Эти методы записывают тестовые данные в базу и, как правило, возвращают добавленные данные, чтобы можно было получить их первичные ключи для использования в тесте;

- своевременно обновляйте методы настройки тестовых данных, осуществляя их рефакторинг при работе с различными сценариями;
- рассмотрите возможность хранения сложных тестовых данных в файле JSON. Я создал метод сериализации данных из рабочей системы в файл JSON, и у меня есть еще один метод, который десериализует эти данные и записывает их в базу данных. Но убедитесь, что вы анонимизируете все персонализированные данные, прежде чем сохранять этот файл;
- метод `EnsureCreated` также заполнит базу данными, настроенными с помощью конфигурации `HasData` (см. раздел 9.4.3).

17.10 Решение проблемы, когда один запрос к базе данных нарушает другой этап теста

В разделе 17.9 описано, как добавлять данные в тестовую базу, прежде чем запускать тест. Это называется *заполнением* базы данных. Но в вашем тесте может возникнуть проблема из-за этапа ссылочной фиксации (см. раздел 6.1.1) в запросе к базе данных. Каждый отслеживаемый запрос к базе данных (то есть запрос без метода `AsNoTracking`) попробует повторно использовать экземпляры всех сущностей, уже отслеживаемых DbContext модульного теста. В результате любой отслеживаемый запрос может повлиять на отслеживаемый запрос после него, что в свою очередь может повлиять на этапы «Действие» и «Проверка».

Лучшие всего объяснить на примере. Предположим, вы хотите протестировать код для добавления нового отзыва в книгу и написали код, показанный в следующем фрагменте:

```
var book = context.Books
    .OrderBy(x => x.BookId).Last();
book.Reviews.Add( new Review{NumStars = 5});
context.SaveChanges();
```

Но здесь есть проблема: в коде ошибка. В первую строку нужно добавить `Include(b => b.Reviews)`, чтобы гарантировать, что сначала загрузятся текущие отзывы. Однако если не проявить осторожность, то ваш модульный тест будет работать так, как показано в следующем листинге.

Листинг 17.20 НЕПРАВИЛЬНАЯ имитация отключенного состояния с неверным результатом

```
[Fact]
public void INCORRECTtestOfDisconnectedState()
{
```

```
        //НАСТРОЙКА
    var options = SqliteInMemory
           .CreateOptions<EfCoreContext>();
    using var context = new EfCoreContext(options);

    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ДЕЙСТВИЕ
    var book = context.Books
        .OrderBy(x => x.BookId).Last();
    book.Reviews.Add(new Review { NumStars = 5 });
    context.SaveChanges();

    //ПРОВЕРКА
    //ЭТО НЕПРАВИЛЬНО!!!!!
    context.Books
       .OrderBy(x => x.BookId).Last()
       .Reviews.Count.ShouldEqual(3);
}
```

Настраивает тестовую базу данных с тестовыми данными, состоящими из четырех книг

Читает последнюю книгу из набора тестов, у которой, как вы знаете, есть два отзыва

Добавляет в книгу еще один отзыв, который не должен сработать, но он работает, потому что исходные данные все еще отслеживаются экземпляром DbContext

Сохраняет отзыв в базе данных

Проверяет наличие трех отзывов. Это работает, но модульный тест должен завершиться неудачно, выбросив исключение

Фактически в этом модульном тесте две ошибки из-за отслеживаемых сущностей:

- этап *Действие* – должен был завершиться неудачей, потому что навигационное свойство `Reviews` было равно `null`, но все работает из-за ссылочной фиксации на этапе *Настройка*;
- этап *Проверка* – должен завершиться ошибкой, если вызов `context.SaveChanges` был пропущен, но все работает из-за ссылочной фиксации на этапе *Действие*.

На мой взгляд, худший результат (который даже хуже, чем отсутствие модульного теста) – это модульный тест, который работает не так, как нужно. Вы думаете, что все хорошо, а на самом деле это не так. Посмотрим, как можно изменить неверный модульный тест из листинга 17.20, чтобы он сработал должным образом. Раньше был только один способ решить эту проблему, но после выхода EF Core 5 появился еще один подход. Вот эти два варианта:

- использовать метод `ChangeTracker.Clear` в EF Core 5, чтобы очистить отслеживаемые сущности;
- использовать несколько экземпляров DbContext в рамках областей видимости `using` (оригинальный подход).

Я считаю, что подход с методом `ChangeTracker.Clear` позволяет быстрее писать код и он короче, поэтому покажу его первым, а также продемонстрирую оригинальный подход с несколькими экземплярами DbContext для сравнения.

17.10.1 Код теста с методом ChangeTracker.Clear в отключенном состоянии

Следующий листинг решает проблему заполнения данными, влияющими на этап «Действие», который влияет на этап проверки. В этом случае генерируется исключение, поскольку коллекция `Reviews` имеет значение `null` (при условии что вы следовали моей рекомендации из раздела 6.1.6). Если этап «Действие» был исправлен, то код на этапе «Проверка» сможет определить, что метод `SaveChanges` не был вызван.

Листинг 17.21 Использование метода `ChangeTracker.Clear` для правильной работы модульного теста

```
[Fact]
public void UsingChangeTrackerClear()
{
    //НАСТРОЙКА
    var options = SqliteInMemory
        .CreateOptions<EfCoreContext>();
    using var context = new EfCoreContext(options);

    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    context.ChangeTracker.Clear();

    //ДЕЙСТВИЕ
    var book = context.Books
        .OrderBy(x => x.BookId).Last();
    book.Reviews.Add(new Review { NumStars = 5 });
    context.SaveChanges();

    //ПРОВЕРКА
    context.ChangeTracker.Clear();

    context.Books.Include(b => b.Reviews)
        .OrderBy(x => x.BookId).Last()
        .Reviews.Count.ShouldEqual(3);
}
```

Настраивает тестовую базу данных с тестовыми данными, состоящими из четырех книг

Вызывает метод ChangeTracker.Clear, чтобы прекратить отслеживание всех сущностей

Читает последнюю книгу из набора тестов, у которой, как вы знаете, есть два отзыва

Сохраняет отзыв в базе данных

Когда вы пытаетесь добавить новый отзыв, EF Core генерирует исключение NullReferenceException, потому что коллекция Review сущности Book не загружена и, следовательно, имеет значение null

Повторно загружает книгу с отзывами, чтобы проверить, нет ли трех отзывов

Вызывает метод ChangeTracker.Clear, чтобы прекратить отслеживание всех сущностей

Если вы сравните листинг 17.21 с листингом 17.22, то увидите, что код стал короче на девять строк, главным образом потому, что вам не нужны все блоки `using` из листинга 17.22. Кроме того, я считаю, что код легче читать без всех этих блоков.

17.10.2 Код теста с несколькими экземплярами DbContext в отключенном состоянии

В следующем листинге используются два экземпляра DbContext: один для настройки базы данных и второй для запуска теста. Тест проходит неудачно, потому что генерируется исключение, поскольку значение коллекции `Reviews` равно `null` (при условии что вы следовали моей рекомендации из раздела 6.1.6).

Листинг 17.22 Три отдельных экземпляра DbContext, обеспечивающих правильную работу теста

```
[Fact]
public void UsingThreeInstancesOfTheDbcontext()
{
    //НАСТРОЙКА
    var options = SqliteInMemory
        .CreateOptions<EfCoreContext>();
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureCreated();
        context.SeedDatabaseFourBooks();
    }
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {
        //ДЕЙСТВИЕ
        var book = context.Books
            .Include(x => x.Reviews)
            .OrderBy(x => x.BookId).Last();
        book.Reviews.Add(new Review { NumStars = 5 });

        context.SaveChanges();
    }
    using (var context = new EfCoreContext(options))
    {
            //ПРОВЕРКА
            context.Books.Include(b => b.Reviews)
                .OrderBy(x => x.BookId).Last()
                .Reviews.Count.ShouldEqual(3);
    }
}
```

Не дает высвободить подключение к SQLite после высвобождения следующего экземпляра DbContext приложения

Создает параметры SQLite в памяти так же, как в предыдущем примере

Создает первый экземпляр DbContext приложения

Настраивает тестовую базу данных с тестовыми данными, состоящими из четырех книг, но на этот раз в отдельном экземпляре DbContext

Читает последнюю книгу из набора тестов, у которой, как вы знаете, есть два отзыва

Вызывает метод SaveChanges для обновления базы данных

Перезагружает книгу с отзывами, чтобы проверить, нет ли трех отзывов

Закрывает последний экземпляр и открывает новый экземпляр DbContext приложения. В новом экземпляре нет отслеживаемых сущностей, которые могли бы повлиять на выполнение теста

Когда вы пытаетесь добавить новый отзыв, EF Core генерирует исключение NullReferenceException, поскольку коллекция Review не загружена и, следовательно, имеет значение null

17.11 Перехват команд, отправляемых в базу данных

Иногда полезно узнать, что делает EF Core, когда обращается к реальной базе данных и EF Core предоставляет несколько способов сделать это. Проверка сообщений журналов EF Core из работающего приложения – один из способов, но, возможно, будет непросто найти нужные записи. Другой, более сфокусированный подход – написание модульных тестов, которые тестируют определенные части запросов EF Core, перехватывая команды SQL, которые EF Core будет использовать для запросов к базе данных.

Журналы EF Core часто содержат команды SQL, а также другую информацию, например предупреждения о возможных проблемах и хронометраж (сколько времени занял доступ к базе данных). Кроме того, даже если вы плохо знаете язык SQL, нетрудно проверить, привели ли внесенные вами изменения в конфигурацию к ожидаемым изменениям в базе данных. В EF Core 5 появились два нововведения, которые значительно упрощают перехват команд базы данных по сравнению с предыдущими версиями:

- расширение параметра `LogTo`, которое упрощает фильтрацию и перехват сообщений журналов EF Core;
- метод `ToQueryString`, генерирующий SQL-код из LINQ-запроса.

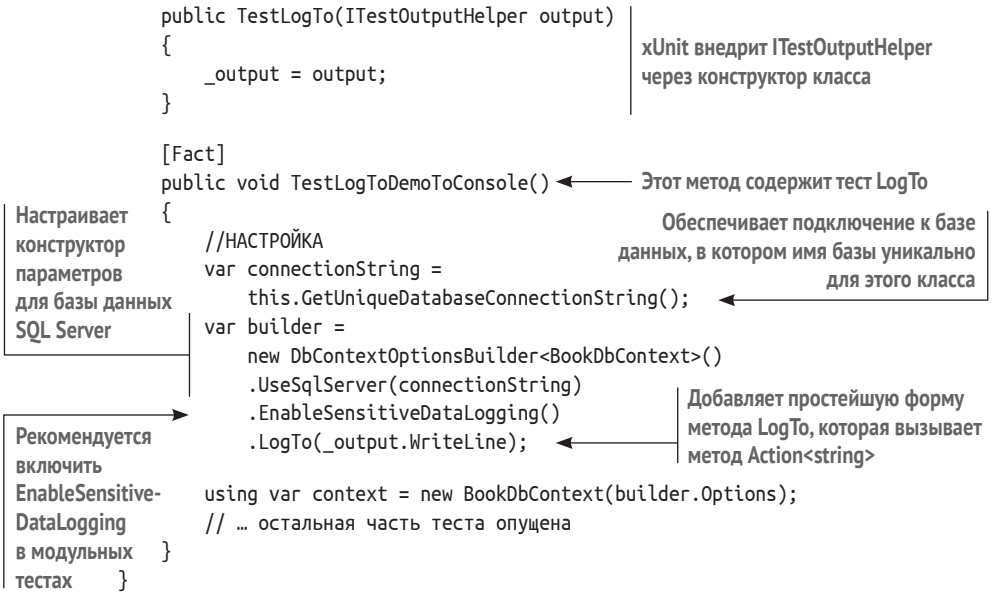
17.11.1 Использование расширения параметра LogTo для фильтрации и перехвата сообщений журналов EF Core

До EF Core 5 для получения журналов из EF Core требовалось создать класс `ILoggerProvider` и зарегистрировать этого поставщика средства ведения журнала с помощью метода расширения `UseLoggerFactory`. Это было непросто. Метод `LogTo` в EF Core 5 значительно упрощает получение сообщений журналов и добавляет возможности для фильтрации журналов, которые вы хотите просмотреть.

Обычно этот метод возвращает каждое сообщение журнала, используя тип `Action<string>`, и вы можете добавлять сообщения журнала в переменную `List<string>` или выводить куда-нибудь в консоль. В xUnit используется метод `WriteLine` интерфейса `ITestOutputHelper`, как показано в следующем листинге.

Листинг 17.23 Вывод сообщений журнала из теста xUnit с использованием метода LogTo

```
public class TestLogTo                    ← Класс, содержащий модульные тесты LogTo
{
    private readonly ITestOutputHelper _output;   ← Интерфейс xUnit, позволяющий выводить
                                                    данные в средство запуска модульных тестов
```



По умолчанию используется следующий формат:

- `LINE1: <loglevel(4 chars)> <DateTime.Now> <EventId> <Category>;`
- `LINE2: <the log message>.`

В следующем фрагменте кода показано одно из сообщений журнала в этом формате:

- `LINE1: warn: 10/12/2020 11:59:38.658 CoreEventId.SensitiveData-Logging-EnabledWarning[10400] (Microsoft.EntityFrameworkCore.Infrastructure);`
- `LINE2: Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.`

Помимо вывода журналов, метод `LogTo` может выполнять фильтрацию по следующим типам:

- `LogLevel`, например `LogLevel.Information` или `LogLevel.Warning`;
- `EventIds`, которые определяют конкретный вывод журнала, например `CoreEventId.ContextInitialized` и `RelationalEventId.CommandExecuted`;
- имена категорий, которые EF Core определяет для команд в группах, например `DbLoggerCategory.Database.Command.Name`;
- функции, которые принимают `EventId` и `LogLevel` и возвращают значение `true` для сообщений журнала, которые вы хотите выводить.

Это великолепный метод, но здесь так много вариантов добавления этой функции в библиотеку EfCore.TestSupport, что я создал класс `LogToOptions` для обработки всех настроек (наряду с кодом для выброса исключения, если выбранная комбинация не поддерживается). Кроме

того, этот класс включает в себя некоторые значения по умолчанию, отличные от значений по умолчанию для `LogTo`, которые основаны на моем опыте журналирования в модульных тестах. Вот эти изменения:

- по умолчанию `LogLevel` должен быть `Information` (я нахожу журналы уровня `Debug` полезными, только если пытаюсь найти ошибку);
- мне не нужен `DateTime` в журнале, потому что это означает, что я не могу сравнить журнал с постоянной строкой, поэтому я установил для параметра `DbContextLoggerOptions` значение `None` (`DbContextLoggerOptions` контролирует вывод журнала и может добавлять дополнительную информацию в строку журнала);
- в большинстве случаев мне не нужны журналы этапа «Настройка» модульного теста, поэтому я добавил свойство `bool ShowLog` (значение по умолчанию – `true`), позволяющее контролировать, когда вызывается параметр `Action<string>`.

Вот листинг с классом `LogToOptions` и комментариями по каждому свойству.

Листинг 17.24 Класс `LogToOptions` со всеми настройками для метода `LogTo`

```
public class LogToOptions
{
    public bool ShowLog { get; set; }
        = true;

    public LogLevel LogLevel { get; set; }
        = LogLevel.Information;

    public string[] OnlyShowTheseCategories
         { get; set; }

    public EventId[] OnlyShowTheseEvents
         { get; set; }

    public Func<EventId, LogLevel, bool>
         FilterFunction { get; set; }

    public DbContextLoggerOptions
         LoggerOptions { get; set; }
             = DbContextLoggerOptions.None
}
```

Будут выводиться только сообщения журнала со значением свойства LogLevel или выше; по умолчанию это LogLevel.Information

Если false, то метод Action<string> не вызывается; по умолчанию true

Если значение не null, возвращает только сообщения журнала с именем Category из этого массива; по умолчанию null

Если значение не null, возвращает только сообщения журнала с EventId из этого массива; по умолчанию null

Управляет форматом журнала EF Core. По умолчанию в журнале не добавляется дополнительная информация, например LogLevel, DateTime и т. д.

Если значение не null, вызывается заданная функция, и возвращаются журналы только в тех случаях, когда эта функция возвращает true; по умолчанию null

Теперь используем класс `LogToOptions` с методом из библиотеки EfCore.TestSupport, `SqliteInMemory.CreateOptionsWithLogTo`. В следую-

щем листинге в классе `LogToOptions` используется свойство `ShowLog`, чтобы сообщения журнала отображались только после завершения этапа «Настройка».

Листинг 17.25 Отключение вывода журнала до завершения этапа //НАСТРОЙКА

```
[Fact]
public void TestEfCoreLoggingCheckSqlOutputShowLog()
{
    //НАСТРОЙКА
    var logToOptions = new LogToOptions
    {
        ShowLog = false
    };
    var options = SqliteInMemory
        .CreateOptionsWithLogTo
        <BookDbContext>(
            _output.WriteLine,
            logToOptions);

    using var context = new BookDbContext(options);
    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ДЕЙСТВИЕ
    logToOptions.ShowLog = true;
    var book = context.Books.Count();

    //ПРОВЕРКА
}
```

В данном случае нужно изменить LogToOptions по умолчанию, чтобы установить для ShowLog значение false

Параметр – это метод Action<string>, и его нужно передать

Этот метод настраивает параметры SQLite in-memory и добавляет к этим параметрам LogTo

Этот раздел настройки и заполнения не дает никаких результатов, потому что свойство ShowLog имеет значение false

Включает вывод сообщений журнала, задав для свойства ShowLog значение true

Второй параметр является необязательным, но в данном случае мы хотим предоставить logToOptions для управления выводом

Этот запрос производит один вывод сообщений журнала, который будет отправлен в окно средства запуска xUnit

В результате, вместо того чтобы пролистывать сообщения журнала из создания и заполнения базы данных, вы видите только один вывод журнала в окне средства запуска xUnit, как показано в следующем фрагменте кода:

```
Executed DbCommand (0ms) [Parameters=[],
        CommandType='Text', CommandTimeout='30']
SELECT COUNT(*)
FROM "Books" AS "b"
WHERE NOT ("b"."SoftDeleted")
```

17.11.2 Использование метода ToQueryString для отображения сгенерированного SQL-кода из LINQ-запроса

С выводом сообщений журналов все отлично, и в них содержится много полезной информации, но если вы просто хотите увидеть, как

выглядит ваш запрос, то есть более простой способ. Если вы создали запрос к базе данных, возвращающий результат `IQueryable`, то можно использовать метод `ToQueryString`. Следующий листинг включает вывод метода `ToQueryString` в тесте.

Листинг 17.26 Модульный тест, содержащий метод `ToQueryString`

```
[Fact]
public void TestToQueryStringOnLinqQuery()
{
    //НАСТРОЙКА
    var options = SqliteInMemory.CreateOptions<BookDbContext>();
    using var context = new BookDbContext(options);
    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ДЕЙСТВИЕ
    var query = context.Books.Select(x => x.BookId);
    var bookIds = query.ToArray();

    //ПРОВЕРКА
    _output.WriteLine(query.ToQueryString());
    query.ToQueryString().ShouldEqual(
        "SELECT \"b\".\"BookId\"\r\n" +
        "FROM \"Books\" AS \"b\"\r\n" +
        "WHERE NOT (\"b\".\"SoftDeleted\")");
    bookIds.ShouldEqual(new []{1,2,3,4});
}
```

Вы предоставляете LINQ-запрос без выполнения

Затем вы выполняете его, добавляя в конце метод ToArray

Выводит SQL-код для LINQ-запроса

Проверяет, соответствует ли SQL-код тому, что вы ожидали

Проверяет вывод запроса

Резюме

- Модульное тестирование – это способ протестировать *модуль* – небольшой фрагмент кода, который может быть логически изолирован в приложении.
- Модульное тестирование – отличный способ выявления ошибок при разработке кода и, что еще важнее, когда вы или кто-то другой осуществляет его рефакторинг.
- Я рекомендую использовать xUnit, потому что он применяется очень широко (EF Core использует xUnit и насчитывает около 70 000 тестов), хорошо поддерживается и быстрый. Кроме того, я создал библиотеку под названием EfCore.TestSupport, которая предоставляет методы, чтобы облегчить тестирование кода EF Core в xUnit.
- DbContext, предназначенный для работы с приложением ASP.NET Core, готов к модульному тестированию, но любой DbContext, использующий метод `OnConfiguring` для установки параметров, необходимо изменить, чтобы разрешить модульное тестирование.
- Есть три основных способа имитации базы данных при модульном тестировании, и у каждого из них свои компромиссы:

  - *использовать тот же тип базы данных, что и рабочая база данных*, – это самый безопасный подход, но нужно разбираться с устаревшими схемами базы данных и управлением базой данных, чтобы разрешить параллельный запуск классов модульного тестирования;
  - *использовать базу данных SQLite in-memory* – это самый быстрый и простой подход, но он не имитирует всю функциональность SQL из рабочей базы данных;
  - *заглушка* – когда у вас есть паттерн «Репозиторий» для доступа к базе данных, например в бизнес-логике (см. раздел 4.4.3), заглушка этого репозитория дает вам быстрый и полный контроль над данными для модульного тестирования, но обычно для этого требуется писать больше кода.
- В Cosmos DB есть удобный эмулятор Azure Cosmos DB, который можно скачать и запустить локально. Это приложение позволяет проводить модульное тестирование Cosmos DB без использования службы Azure Cosmos DB.
- Многие модульные тесты нуждаются в тестовой базе данных, содержащей данные, которые будут использоваться при тестировании, поэтому стоит потратить время на разработку набора методов тестирования, которые создадут тестовые данные для использования в модульных тестах.
- Модульные тесты могут утверждать, что тестируемый код верен, когда это не так. Подобная ситуация может произойти, если один из разделов модульного теста использует отслеживаемые экземпляры из предыдущего этапа теста. Есть два способа убедиться, что такая проблема не возникает: используйте отдельные экземпляры DbContext или метод `ChangeChanger.Clear`.
- В EF Core 5 добавлены два метода, которые значительно упрощают перехват SQL, созданного из кода: `LogTo` для записи выходных данных журнала и метод `ToQueryString` для преобразования запросов LINQ в команды базы данных.

Приложение A
Краткое введение в LINQ

Это приложение предназначено для тех, кто незнаком с проектом Microsoft, Language Integrated Query (LINQ) или кто хочет по-быстрому узнать, как работает LINQ. Язык LINQ устраняет разрыв между миром объектов и миром данных и используется EF Core для построения запросов к базе данных. Понимание этого языка – ключ к использованию EF Core для доступа к базе данных.

Это приложение начинается с описания двух вариантов синтаксиса, которые можно использовать для написания кода LINQ. Кроме того, вы узнаете о типах команд, доступных в LINQ, с примерами того, как эти команды могут управлять коллекциями данных в памяти.

После этого вы познакомитесь со связанным типом .NET `IQueryable<T>`, который содержит код LINQ в форме, который поддерживает отложенное выполнение. Этот тип позволяет разработчикам разбивать сложные запросы на отдельные части и динамически изменять запрос, а также дает возможность EF Core транслировать код LINQ в команды, которые могут быть выполнены на сервере базы данных. В конце вы узнаете, как выглядит запрос EF Core с LINQ.

A1 Введение в язык LINQ

Можно управлять коллекциями данных, используя методы LINQ для сортировки, фильтрации, выбора и т. д. Эти коллекции могут быть данными в памяти (например, массивом целых чисел, данными в формате XML или JSON) и, конечно же, базами данных, используя библиотеки, такие как EF Core. LINQ доступен в языках Microsoft C#, F# и Visual Basic; вы можете создавать читаемый код, используя функциональное программирование.

СОВЕТ Если вы еще не сталкивались с функциональным программированием, стоит познакомиться с ним. Посетите страницу http://mng.bz/97CY или, чтобы получить более подробную информацию, обратитесь к книге Энрико Буонанно «*Функциональное программирование на C#*» (http://mng.bz/Q2Qv).

A 1.1 Два способа написания LINQ-запросов

В LINQ есть два варианта синтаксиса для написания запросов: синтаксис *методов* и синтаксис *запросов*. В этом разделе представлены оба варианта и указано, какой из них здесь используется. Мы напишем один и тот же LINQ-запрос для фильтрации и сортировки массива целых чисел, используя оба варианта.

В листинге A.1 используется т. н. синтаксис *методов*, или *лямбда-выражений*. Этот код состоит из простых операторов LINQ. Даже если вы раньше не видели LINQ, имена методов, такие как `Where` и `OrderBy`, дают хорошее представление о том, что они делают.

Листинг A.1 Наш первый взгляд на LINQ с использованием синтаксиса методов / лямбд

```
int[] nums = new[] {1, 5, 4, 2, 3, 0};

int[] result = nums
    .Where(x => x > 3)
    .OrderBy(x => x)
    .ToArray();
```

Создает массив целых чисел от 0 до 5, но в случайном порядке

Применяет команды LINQ и возвращает новый массив целых чисел

Оставляет все целые числа больше 3

Упорядочивает числа

Преобразует запрос обратно в массив. Результат – массив целых чисел {4, 5}

Название *лямбда* происходит от синтаксиса лямбда-выражений, появившегося в C# 3. Он позволяет писать метод без стандартного синтаксиса определения метода. Фрагмент `x => x > 3` внутри метода `Where` эквивалентен следующему методу:

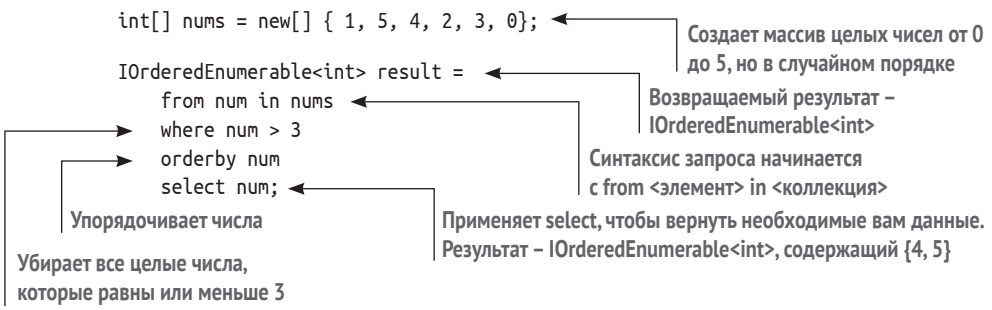
```
private bool AnonymousFunc(int x)
{
    return x > 3;
}
```

Как видите, синтаксис лямбда-выражений позволяет значительно сэкономить на вводе текста. Я использую их во всех моих запросах EF Core и в другом коде, который писал для этой книги.

В следующем листинге показан другой способ написания кода LINQ, называемый синтаксисом *запросов*. Этот код дает тот же результат, что и листинг A.1, но возвращает немного иной тип.

Листинг A.2 Наш первый взгляд на LINQ с использованием синтаксиса запросов

```
int[] nums = new[] { 1, 5, 4, 2, 3, 0};

IOrderedEnumerable<int> result =
    from num in nums
    where num > 3
    orderby num
    select num;
```

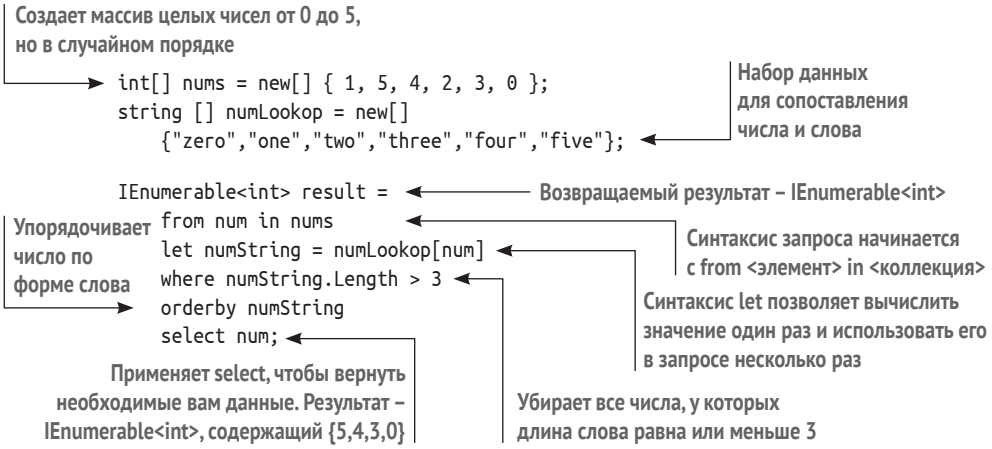


Вы можете использовать любой синтаксис; выбор за вами. Я использую синтаксис методов, потому что он требует меньшего набора текста на клавиатуре и потому что мне нравится способ объединения команд в цепочку, одна за другой. В остальных примерах книги используется именно этот синтаксис.

Прежде чем оставить тему синтаксиса, хочу представить вам концепцию предварительного вычисления значения в запросе LINQ. В синтаксисе запросов есть функциональность, специально предназначенная для этой задачи: ключевое слово `let`. Оно позволяет вычислить значение один раз, а затем использовать его несколько раз в запросе, чтобы сделать запрос более эффективным. В этом листинге показан код, преобразующий целочисленное значение в словесный/строковый эквивалент, а затем использующий эту строку как часть запроса сортировки и фильтрации.

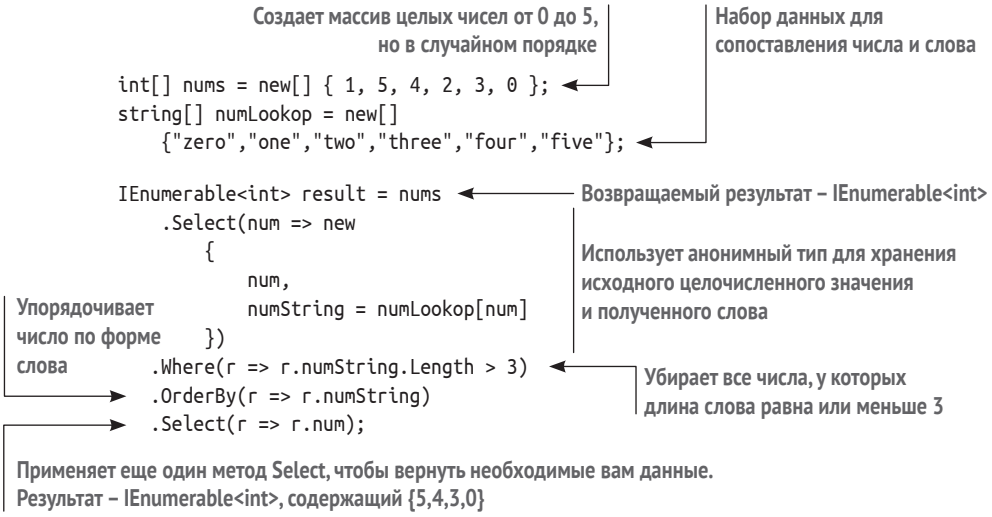
Листинг A.3 Использование ключевого слова `let` в синтаксисе запросов LINQ

```
int[] nums = new[] { 1, 5, 4, 2, 3, 0 };
string [] numLookop = new[]
    {"zero","one","two","three","four","five"};

IEnumerable<int> result =
    from num in nums
    let numString = numLookop[num]
    where numString.Length > 3
    orderby numString
    select num;
```



Эквивалент в синтаксисе методов – это оператор `Select`, как показано в следующем листинге (в разделе А.1.2 содержатся дополнительные сведения об этом операторе).

Листинг А.4 Использование оператора `Select` для хранения вычисленного значения



> **EF6** В EF6.x ключевое слово `let` или метод `Select` использовались как подсказки для предварительного вычисления значения в базе только один раз. В EF Core нет такой оптимизации производительности, поэтому он пересчитывает значение при каждом вхождении.

А 1.2 Операции с данными, которые можно выполнять в LINQ

В LINQ есть множество методов, называемых *операторами*. У большинства операторов есть имена и функции, которые четко указывают на то, что они делают. В табл. А.1 перечислены некоторые наиболее распространенные операторы LINQ; похожие операторы сгруппированы, чтобы можно было увидеть, где их использовать. Данный список не является исчерпывающим; его цель – показать вам самые распространенные операторы, чтобы вы прочувствовали, на что способен LINQ.

Таблица А.1 Примеры операторов LINQ, сгруппированных по назначению

Группа	Примеры (показаны не все операторы)
Сортировка	`OrderBy`, `OrderByDescending`, `Reverse`
Фильтрация	`Where`
Выбор элемента	`First`, `FirstOrDefault`

Таблица А.1 (*окончание*)

Группа	Примеры (показаны не все операторы)
Проекция	`Select`
Агрегация	`Max, Min, Sum, Count, Average`
Разбиение	`Skip, Take`
Логическая проверка	`Any, All, Contains`

В листинге А.4 показан запрос LINQ, сортирующий и фильтрующий массив целых чисел. Теперь мы рассмотрим несколько примеров, в которых запрос LINQ работает с классом C#. Во-первых, нужно определить новый класс `Review` с данными, который потребуется нам в примерах, как показано в следующем листинге.

Листинг А.5 Класс `Review` и переменная `ReviewsList`, содержащая два отзыва

```
class Review
{
    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }
}

List<Review> ReviewsList = new List<Review>
{
    new Review
    {
        VoterName = "Jack",
        NumStars = 5,
        Comment = "A great book!"
    },
    new Review
    {
       VoterName = "Jill",
       NumStars = 1,
       Comment = "I hated it!"
    }
};
```

Поле `ReviewsList` показано в табл. А.2, которая должна дать вам представление о том, как работают различные операторы LINQ.

Таблица А.2 Четыре варианта использования LINQ с полем класса `ReviewsList` в качестве данных. Результат каждого оператора отображается в столбце «Результат»

Группа	Код с использованием операторов LINQ	Результат

Проекция	`string[] result = ReviewsList`
	`.Select(p => p.VoterName)`
	`.ToArray();`
		`string[]{"Jack", "Jill"}`

Таблица A.2 (*окончание*)

Группа	Код с использованием операторов LINQ	Результат
Агрегация	`double result = ReviewsList .Average(p => p.NumStars);`	`3 (среднее между 5 и 1)`
Выбор элемента	`string result = ReviewsList .First().VoterName;`	`"Jack" (первый отзыв)`
Логическая проверка	`bool result = ReviewsList .Any(p => p.NumStars == 1);`	`true (Jill поставила 1)`

A.2 Введение в тип IQueryable<T> и почему он полезен

Еще одна важная часть LINQ – обобщенный интерфейс `IQueryable<T>`. LINQ – довольно особенный язык, поскольку любой набор его операторов (для EF Core), который вы предоставляете, не выполняется сразу же, а хранится в типе `IQueryable<T>`, ожидая команды для материализации запроса. У такой формы `IQueryable<T>` есть два преимущества:

- с помощью типа `IQueryable<T>` можно разделить сложный запрос LINQ на отдельные части;
- EF Core может транслировать `IQueryable<T>` в команды запросов к базе данных.

A.2.1 Разделение сложного запроса LINQ с помощью типа IQueryable<T>

В этой книге вы познакомились с объектами запроса (см. раздел 2.6) и создали сложный запрос для вывода списка книг путем объединения трех объектов запроса. Эта операция удалась благодаря способности типа `IQueryable<T>` хранить код в специальной форме, которая называется *деревом выражения*, так чтобы к нему можно было добавлять другие операторы LINQ.

В качестве примера мы улучшим код из листинга A.1, добавив собственный метод, содержащий часть запроса, позволяющую изменять порядок сортировки запроса LINQ. Мы создадим его как метод расширения, что позволит встроить его в цепочку методов так же, как это делают операторы LINQ. (Операторы LINQ – это методы расширения.)

> **ОПРЕДЕЛЕНИЕ** *Метод расширения* – это статический метод в статическом классе, перед первым параметром которого стоит ключевое слово `this`. Чтобы разрешить связывание методов в цепочку, метод должен также возвращать тип, который другие методы могут использовать в качестве входных данных.

В листинге А.6 показан метод расширения `MyOrder`, принимающий тип `IQueryable<int>` в качестве первого параметра и возвращающий результат `IQueryable<int>`. Кроме того, здесь есть второй логический параметр `ascending`, устанавливающий порядок сортировки по возрастанию или по убыванию.

Листинг А.6 Наш метод инкапсулирует часть кода LINQ, используя `IQueryable<int>`

Предоставляет второй параметр, позволяющий изменить порядок сортировки

Метод расширения должен быть определен в статическом классе

Статический метод MyOrder возвращает IQueryable<int>, поэтому другие методы расширения могут быть объединены в цепочку

Первый параметр метода расширения – IQueryable, и ему предшествует ключевое слово this

Использует логический параметр ascending, чтобы указать, добавляете вы LINQ-оператор OrderBy или OrderByDescending к результату IQueryable

Параметр ascending имеет значение false, поэтому к параметру IQueryable добавляется оператор OrderByDescending

Параметр ascending имеет значение true, поэтому к параметру IQueryable добавляется оператор OrderBy

```
public static class LinqHelpers
{
    public static IQueryable<int> MyOrder
        (this IQueryable<int> queryable,
        bool ascending)
    {
        return ascending
            ? queryable
                .OrderBy(num => num)
            : queryable
                .OrderByDescending(num => num);
    }
}
```

В этом листинге используется наш метод расширения с `IQueryable<int>` для замены оператора LINQ `OrderBy` в исходном коде листинга А.1.

Листинг А.7 Использование метода `MyOrder` с `IQueryable<int>` в коде LINQ

Преобразует массив целых чисел в объект IQueryable

Вызывает метод MyOrder IQueryable<int> со значением true, устанавливая сортировку данных по возрастанию

Выполняет IQueryable и преобразует результат в массив. Результат – массив целых чисел {4, 5}

Убирает все числа, которые равны или меньше 3

```
var numsQ = new[] { 1, 5, 4, 2, 3 }
    .AsQueryable();

var result = numsQ
    .MyOrder(true)
    .Where(x => x > 3)
    .ToArray();
```

Такие методы расширения, как `MyOrder`, предоставляют две полезные возможности:

- *делают код LINQ динамичным*. Изменяя параметр в `MyOrder`, можно изменить порядок сортировки запроса LINQ. Если бы у вас не

было этого параметра, вам бы понадобились два запроса: один с использованием `OrderBy` и второй с `OrderByDescending` – а затем нужно было бы выбрать, какой из них вы хотите выполнить, с помощью оператора `if`. Такой подход – плохая практика при разработке программного обеспечения, так как вы будете дублировать код LINQ, например часть с `Where`;

- *позволяют разделить сложные запросы на серию отдельных методов расширения, которые можно объединить в цепочку*. Такой подход упрощает построение, тестирование и понимание сложных запросов. В разделе 2.9 мы разбиваем довольно сложный запрос на вывод списка книг в приложении Book App на отдельные *объекты запроса*. В следующем листинге этот процесс показан снова, а каждый объект запроса выделен жирным шрифтом.

Листинг A.8 Запрос на вывод списка книг с объектами запроса select, order, filter и page

```
public IQueryable<BookListDto> SortFilterPage
    (SortFilterPageOptions options)
{
    var booksQuery = _context.Books
        .AsNoTracking()
        .MapBookToDto()
        .OrderBooksBy(options.OrderByOptions)
        .FilterBooksBy(options.FilterBy, options.FilterValue);

    options.SetupRestOfDto(booksQuery);

    return booksQuery.Page(options.PageNum-1, options.PageSize);
}
```

В запросе используются обе упомянутые мною возможности: их применение позволяет динамически изменять сортировку, фильтрацию и разбиение на страницы списка книг, а также скрывает часть сложного кода за метко названным методом, который сообщает вам, что он делает.

A.2.2 Как EF Core транслирует IQueryable<T> в код базы данных

EF Core транслирует код LINQ в код базы данных, который может выполняться на сервере базы данных. Такое возможно, потому что тип `IQueryable<T>` содержит весь код LINQ в виде дерева выражений, которое EF Core может преобразовать в код доступа к базе данных. На рис. A.1 показано, что EF Core делает за кулисами, когда транслирует запрос LINQ в код доступа к базе данных.

EF Core предоставляет множество дополнительных методов для расширения доступных операторов LINQ. Методы EF Core добавляются в дерево выражений, например `Include`, `ThenInclude` (см. раз-

дел 2.4.1) и т. д. Другие методы EF предоставляют асинхронные версии (см. раздел 5.10) методов LINQ, например `ToListAsync` и `LastAsync`.

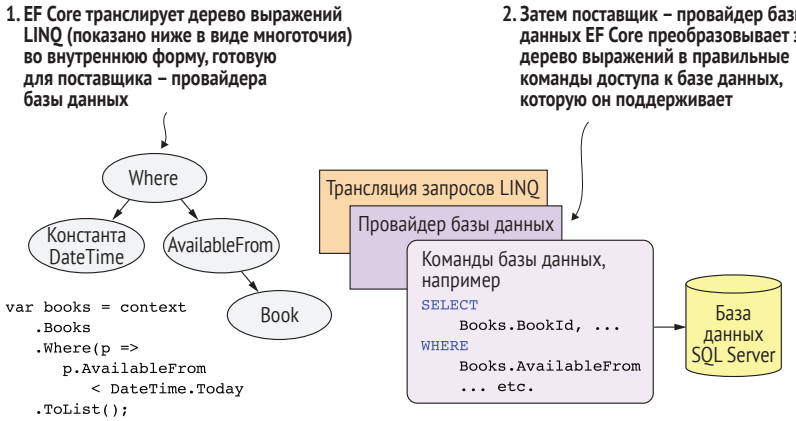


Рис. A.1 Код запроса книги (внизу слева) с деревом выражений над ним. EF Core выполняет два этапа преобразования дерева, прежде чем оно будет преобразовано в форму, подходящую для базы данных, на которую ориентировано приложение

A.3 Запросы к базе данных в EF Core с помощью LINQ

Использование LINQ в запросе к базе данных в EF Core требует трех частей, показанных на рис. A.2. Запрос полагается на DbContext приложения, который описан в разделе 2.2.1. В этом разделе рассматривается только формат запроса к базе данных EF Core. Операторы LINQ выделены жирным шрифтом.

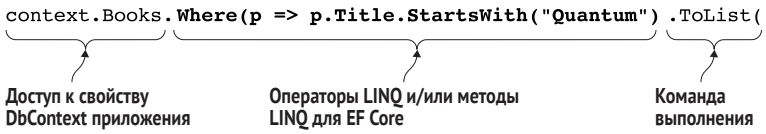


Рис. A.2 Пример доступа к базе данных с тремя частями

Вот как выглядят эти три части запроса к базе данных EF Core:

- *доступ к свойству DbContext* – в DbContext вы определяете свойство с помощью типа `DbSet<T>`. Этот тип возвращает источник данных `IQueryable<T>`, к которому можно добавить операторы LINQ для создания запроса к базе данных;
- *операторы LINQ и/или LINQ-методы для EF Core* – здесь идет код LINQ-запроса к базе данных;

- *команда материализации/выполнения* – такие команды, как `ToList` и `First`, запускают EF Core для преобразования команд LINQ в команды доступа к базе данных, которые выполняются на сервере базы данных.

В главе 2 и далее вы увидите гораздо более сложные запросы, но все они состоят из трех частей, показанных на рис. А.2.

Это контрольный список проблем, связанных с производительностью EF Core, где указан раздел книги, посвященный каждой проблеме.

В следующей таблице представлен список тем, о которых идет речь в этой книге, со списком глав, в которых рассматривается каждая тема, при этом основная глава указана первой. Кроме того, в ней перечислены все ключевые рисунки, относящиеся к этой теме.

Темы	Главы	Ключевые рисунки
Настройка EF Core	1, 2, 7, 8, 5	1.4, 2.7, 1.5
Запрос к базе данных	2, 5, 6	2.7, 2.8, 6.2
Create, Update, Delete	3, 5, 8, 11	3.1, 3.2, с 6.8 по 10
Внутреннее устройство EF Core	1, 6	1.6, 1.8, 1.10, с 6.8 по 6.10
Бизнес-логика	4, 5, 13	4.2, 4.4, 5.4
ASP.NET Core	5, 2	5.1, 5.4
Внедрение зависимостей	5, 14, 15	5.2, 5.3
Async/await	5, 14	5.8, 5.9
Настройка нереляционных свойств	7	7.1, 7.2
Настройка связей	8	8.1, 8.2, 8.3, 8.4
Настройка отображений в таблицах	8	8.12, 8.13
Миграции баз данных	9, 5	9.2, 9.3, 9.5, 9.7, 9.8
Проблемы параллельного доступа	10, 15	10.3, 10.4, 10.5, 10.6, 10.7, 15.7
Использование чистого SQL	11, 6	15.3
Предметно-ориентированное проектирование	13, 4	13.3, 4.2, 13.4, 13.5
Настройка производительности	14, 15, 16	14.1, 15.2, 15.4, 15.9, 16.7, 16.8
Cosmos DB и другие базы данных	16, 17	16.1, 16.3, 16.4, 17.5
Валидация данных	4, 7, 10	
Модульное тестирование	17	17.2, 17.3
Язык LINQ	Приложение, 2	A.2, A.1

	Раздел
Проанализируйте свою производительность	
Правильно ли вы выбрали функции для настройки производительности?	14.1.2
Вы диагностировали проблему, связанную с производительностью?	14.2
Вы оценивали пользовательский опыт?	14.2.1
Проверяли ли вы генерируемый код SQL на предмет низкой производительности?	14.2.3
Запрос к базе данных	
Вы загружаете слишком много столбцов?	14.4.1
Вы загружаете слишком много строк?	14.4.2
Вы используете отложенную загрузку?	14.4.3
Вы сообщаете EF Core, что ваш запрос с доступом только на чтение?	14.4.4
Вы делаете слишком много обращений к базе данных?	14.5.1
Вы добавили индексы к свойствам, которые сортируете или фильтруете?	14.5.2
Вы используете самый быстрый способ загрузки сущности?	14.5.3
Выполняется ли часть запроса на стороне приложения?	14.5.4
Можно ли перенести какие-то вычисления в базу данных?	14.5.5
Заменили ли вы какой-либо неоптимальный SQL-код, созданный LINQ-запросом?	14.5.6
Можете ли вы предварительно скомпилировать часто используемые запросы?	14.5.7
Запись в базу данных	
Вы вызываете метод `SaveChanges` несколько раз?	14.6.1
Вы слишком интенсивно работаете с `DetectChanges`?	14.6.2
Вы использовали `HashSet<T>` для навигационных свойств?	14.6.3
Вы вызываете метод `Update`, когда в этом нет необходимости?	14.6.4
Масштабируемость приложения	
Вы используете пул DbContext?	14.7.1
Вы используете паттерн async/await во всем приложении?	14.7.2
Вы выбрали правильную архитектуру для обеспечения высокой масштабируемости?	14.7.4
Рассматривали ли вы возможность использования Cosmos DB в качестве внешнего кеша?	16.3

Предметный указатель

Латиница

Async/await, 206
BookApp, слой, 93
Camel case, соглашение, 281
cookie-корзина, 156
DataLayer, слой, 93
Data Transfer Object (DTO), тип класса, 88
DbContext, создание, 72
Entity Framework Core (EF Core), 36
ForeignKey, аннотация, 300
InverseProperty, аннотация, 301
JSON Patch, 519
ServiceLayer, слой, 93

А

Автомасштабирование, 177
Агрегат, 506
Альтернативный ключ, 318
Анемичная модель предметной области, 144
Аннотации данных, 260
Архитектура микросервисов, 496
Асинхронное программирование, 206
Атомарная единица, 57

Б

Бизнес-логика, 140
Бизнес-правило, 140
Большой комок грязи, 525

В

Валидация, 141, 148
Вложенность, 606, 621
Внедрение
  зависимостей (dependency injection, DI), 177
  через конструктор, 180
  через параметр, 193
Время простоя, 376
Выборочная загрузка, 82
Выпуск с долгосрочной поддержкой (LTS), 57
Вычисление на стороне клиента, 85
Вычисляемый столбец, 390

Г

Генератор значений, 397
Генерируемый столбец, 389
Гибридные сущности DDD, 519

Глобальные фильтры запросов, 274
Глобальный уникальный идентификатор, 401
Горизонтальное масштабирование, 177, 202

Д

Денормализация, 573
Дерево выражений, 77
Дерево выражения, 680
Динамический вычисляемый столбец, 390
Диспетчер событий, 465

Е

Единица работы (Unit Of Work), 108
Единицы запроса, 616

Ж

Жизненный цикл, 180

З

Зависимая сущность, 118
Заглушка, 153
Задача, 210
Заполнение базы данных, 205, 665
Запрос с доступом на чтение и запись, 77

И

Иерархические данные, 68, 220
Имитация, 153
Индекс, 271
  SQL, 256
Инициализаторы базы данных, 205
Источник
  истины, 343
  событий, 403

К

Каскадное удаление, 136
Кеширование, 573
Класс сущности, 70, 257
Комок грязи (код), 495
Компаратор значений, 268
Контейнер свойств, 335
Контроллер, 187
Конфликты параллельного доступа, 402
Корневой объект, 506

М

Масштабируемость, 591
  приложения, 531
Метод расширения, 91, 680
Методы
  действий, 187
  доступа, 502
Миграция базы данных, 250
«Многие ко многим» (many-to-many), 66, 307
Многослойная архитектура, 92
Моделирование базы данных, 50
Модель предметной области, 145
Модель размещения Blazor Server, 182
Мультитенантная система (multitenant system), 226
Мягкое удаление, 133

Н

Неизменяемый список Errors, 148
Некритическое изменение, 342
Немедленная загрузка, 78
Необязательная зависимая связь, 137
Необязательная связь, 298
Нормализация, 600

О

Обработчик событий, 465
Обратная миграция, 365
Обратное проектирование (Reverse engineering), 71
Объект-значение, 321
Объектно-реляционное несоответствие, 41
Обычный запрос, 77

Обязательная связь, 297
Ограниченные контексты, 498
«Один к одному» (one-to-one), 64, 303
«Один ко многим» (one-to-many), 65, 306
Одиночка (singleton), 181
Ожидания пользователя, 532
Определение тела выражения, 344
Основная сущность, 118
Отключенный тип обновления, 111
Отложенная загрузка, 83
Отслеживаемая сущность, 103, 421
Отслеживаемый запрос, 57

П

Паттерн
  «Адаптер», 152
  Модель–представление–контроллер (MVC), 176
  Объект запроса, 544
Полностью определенная связь, 296
Пользовательская функция SQL, 383
Поставщик журналов, 538
Постоянный вычисляемый столбец, 390
Примитивные типы .NET, 258
Проекция, 600, 605
Пропускная способность, 531
Процедурный паттерн бизнес-логики, 144

Р

Рабочая база данных, 643
Разбиение на страницы, 542
Разбиение таблицы, 333
Развертывание приложения на хосте, 199
Разделение ответственностей (Separation of Concerns, SoC), 187
Разделение таблицы (Table splitting), 68
Регистрация сервиса, 179
Резервное поле, 278

С

Сглаживание, 232
Сегментирование базы данных, 456
Сервис, 180
Скалярные пользовательские функции, 383
Скалярные свойства, 252
Сложная бизнес-логика, 142
Слой представления, 93
«Сначала база данных», подход, 45
«Сначала код», подход, 45
«Сначала проектирование», подход, 46
Снимок отслеживания изменений данных, 54
Собственные типы, 320
Собственный тип (Owned Type class), 68
Событие
  интеграции, 467
  предметной области, 465
  сущности, 464
Соглашение Camel case, 281
Сопоставление (collation), 96
Составной ключ, 128, 260, 269
Ссылочная фиксация (relational fixup), 216
Ссылочная целостность, 135
Строка подключения, 179, 183
Схема базы данных, 273, 339
Сценарий
  изменения SQL, 360
  транзакции, 144

Т

Таблица на иерархию (TPH), 68
Таблица на тип (TPT), 68
Табличные пользовательские функции, 383
Текучий интерфейс (fluent interface), 76
Теневые свойства, 276, 298
Транзакционный BizRunner, 169

Транзакция, 57, 108, 169

Ф

Фильтрация, 543
Фильтры запросов, 274

Х

Хранимые процедуры SQL, 383
Хранимый генерируемый столбец, 390

Ц

Цепочка, 100

Ш

Шардинг, 559

Я

Явная загрузка, 81

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Джон П. Смит

Entity Framework Core в действии

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Беликова Д. А.*
Научный редактор *Кулаков А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 56,06. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Entity Framework Core в действии

Entity Framework радикально упрощает доступ к данным в приложениях .NET. Этот простой в использовании инструмент объектно-реляционного отображения (ORM) позволяет писать код базы данных на чистом C#. Он автоматически отображает классы в таблицы базы данных, разрешает запросы со стандартными командами LINQ и даже генерирует SQL-код за вас!

Данная книга научит вас писать код для беспрепятственного взаимодействия с базой данных при работе с приложениями .NET. Следуя соответствующим примерам из обширного опыта автора книги, Джона Смита, вы быстро перейдете от основ к продвинутым методам. Помимо новейших функциональных возможностей EF, в этой книге рассматриваются вопросы производительности, безопасности, рефакторинга и модульного тестирования.

Издание предназначено для разработчиков .NET, знакомых с реляционными базами данных.

«Наиболее полный справочник по EF Core, который есть или когда-либо будет».
Стивен Бирн, Intel Corporation

«Исчерпывающее руководство по EF Core. Это самый практичный способ улучшить свои навыки по работе с EF Core на примерах из реальной жизни».
Пол Браун, Diversified Services Network

«Я твердо верю, что любой, кто использует EF Core, найдет в этой книге что-то ценное для себя».
Энн Эпштейн, Headspring

«Для меня книга служит полезным ресурсом при работе с Entity Framework».
Фостер Хейнс, J2 Interactive

Рассматриваемые темы:

- настройка EF для определения каждой таблицы и столбца;
- обновление схемы по мере роста приложения;
- интеграция EF с существующим приложением C#;
- написание и тестирование кода бизнес-логики для доступа к базе данных;
- применение предметно-ориентированного проектирования к EF Core;
- получение максимальной производительности от EF Core;
- нереляционные базы данных.

Джон П. Смит – внештатный разработчик программного обеспечения и архитектор, специализирующийся на .NET и Azure.

Интернет-магазин: www.dmkpress.com
Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-93700-114-6