



Язык программирования C# 9 и платформа .NET 5

ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ ПРОГРАММИРОВАНИЯ

10-Е ИЗДАНИЕ

Эндрю Троелсен, Филипп Джепикс

Язык программирования C# 9 и платформа .NET 5:

**ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ
ПРОГРАММИРОВАНИЯ**

10-е издание

2 том

Pro C# 9 with .NET 5

**FOUNDATIONAL PRINCIPLES AND PRACTICES
IN PROGRAMMING**

Tenth Edition

Andrew Troelsen
Philip Japikse

Apress®

Язык программирования C# 9 и платформа .NET 5:

**ОСНОВНЫЕ ПРИНЦИПЫ И ПРАКТИКИ
ПРОГРАММИРОВАНИЯ**

10-е издание

2 ТОМ

**Эндрю Троелсен
Филипп Джепикс**

Київ
Комп'ютерне видавництво
“ДІАЛЕКТИКА”
2022

УДК 004.432

Т70

Перевод с английского и редакция Ю.Н. Артеменко

Троелсен, Э., Джепикс, Ф.

Т70 Язык программирования C# 9 и платформа .NET 5: основные принципы и практики программирования, том 2, 10-е изд./Эндрю Троелсен, Филипп Джепикс; пер. с англ. Ю.Н. Артеменко. — Киев : “Диалектика”, 2022. — 632 с. : ил. — Парал. тит. англ.

ISBN 978-617-7987-82-5 (укр., том 2)

ISBN 978-617-7987-80-1 (укр., многотом.)

ISBN 978-1-4842-6938-1 (англ.)

В 10-м издании книги описаны новейшие возможности языка C# 9 и .NET 5 вместе с подробным “закулисным” обсуждением, призванным расширить на- выки критического мышления разработчиков, когда речь идет об их ремесле. Книга охватывает ASP.NET Core, Entity Framework Core и многое другое наряду с последними обновлениями унифицированной платформы .NET, начиная с улучшений показателей производительности настольных приложений Windows в .NET 5 и обновления инструментария XAML и заканчивая расширенным рас- смотрением файлов данных и способов обработки данных. Все примеры кода были переписаны с учетом возможностей последнего выпуска C# 9.

УДК 004.432

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми мар- ками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Apress Media, LLC.

Copyright © 2021 by Andrew Troelsen, Phillip Japikse.

All rights reserved.

Authorized translation from the *Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming* (ISBN 978-1-4842-6938-1), published by Apress Media, LLC.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher.

ISBN 978-617-7987-82-5 (укр., том 2)

ISBN 978-617-7987-80-1 (укр., многотом.)

ISBN 978-1-4842-6938-1 (англ.)

© “Диалектика”, перевод, 2022

© 2021 by Andrew Troelsen, Phillip Japikse

ОГЛАВЛЕНИЕ

Часть VI. Работа с файлами, сериализация объектов и доступ к данным	19
Глава 20. Файловый ввод-вывод и сериализация объектов	20
Глава 21. Доступ к данным с помощью ADO.NET	61
Часть VII. Entity Framework Core	125
Глава 22. Введение в Entity Framework Core	126
Глава 23. Построение уровня доступа к данным с помощью Entity Framework Core	183
Часть VIII. Разработка клиентских приложений для Windows	265
Глава 24. Введение в Windows Presentation Foundation и XAML	266
Глава 25. Элементы управления, компоновки, события и привязка данных в WPF	302
Глава 26. Службы визуализации графики WPF	363
Глава 27. Ресурсы, анимация, стили и шаблоны WPF	399
Глава 28. Уведомления WPF, проверка достоверности, команды и MVVM	439
Часть IX. ASP.NET Core	477
Глава 29. Введение в ASP.NET Core	478
Глава 30. Создание служб REST с помощью ASP.NET Core	530
Глава 31. Создание приложений MVC с помощью ASP.NET Core	555
Предметный указатель	627

СОДЕРЖАНИЕ

Часть VI. Работа с файлами, сериализация объектов и доступ к данным	19
Глава 20. Файловый ввод-вывод и сериализация объектов	20
Исследование пространства имен System.IO	20
Классы Directory (DirectoryInfo) и File (FileInfo)	22
Абстрактный базовый класс FileSystemInfo	22
Работа с типом DirectoryInfo	23
Перечисление файлов с помощью типа DirectoryInfo	25
Создание подкаталогов с помощью типа DirectoryInfo	25
Работа с типом Directory	26
Работа с типом DirectoryInfo	27
Работа с типом FileInfo	28
Метод FileInfo.Create()	29
Метод FileInfo.Open()	30
Методы FileInfo.OpenRead() и FileInfo.OpenWrite()	31
Метод FileInfo.OpenText()	32
Методы FileInfo.CreateText() и FileInfo.AppendText()	32
Работа с типом File	33
Дополнительные члены типа File	34
Абстрактный класс Stream	35
Работа с типом FileStream	36
Работа с типами StreamWriter и StreamReader	37
Запись в текстовый файл	38
Чтение из текстового файла	39
Прямое создание объектов типа StreamWriter/StreamReader	39
Работа с типами StringWriter и StreamReader	40
Работа с типами BinaryWriter и BinaryReader	41
Программное слежение за файлами	43
Понятие сериализации объектов	45
Роль графов объектов	46
Создание примеров типов и написание операторов верхнего уровня	47
Сериализация и десериализация с помощью XmlSerializer	49
Сериализация и десериализация с помощью System.Text.Json	53
Резюме	60
Глава 21. Доступ к данным с помощью ADO.NET	61
Сравнение ADO.NET и ADO	62
Поставщики данных ADO.NET	62
Поставщики данных ADO.NET	64
Типы из пространства имен System.Data	64
Роль интерфейса IDbConnection	65

Роль интерфейса <code>IDbTransaction</code>	66
Роль интерфейса <code>IDbCommand</code>	66
Роль интерфейсов <code>IDbDataParameter</code> и <code>IDataParameter</code>	66
Роль интерфейсов <code>IDbDataAdapter</code> и <code>IDataAdapter</code>	67
Роль интерфейсов <code>IDataReader</code> и <code>IDataRecord</code>	68
Абстрагирование поставщиков данных с использованием интерфейсов	69
Установка SQL Server и Azure Data Studio	71
Установка SQL Server	72
Установка IDE-среды SQL Server	74
Подключение к SQL Server	74
Восстановление базы данных <code>AutoLot</code>	76
из резервной копии	76
Копирование файла резервной копии в имеющийся контейнер	76
Восстановление базы данных с помощью SSMS	77
Восстановление базы данных с помощью Azure Data Studio	78
Создание базы данных <code>AutoLot</code>	78
Создание базы данных	79
Создание таблиц	79
Создание отношений между таблицами	81
Создание хранимой процедуры <code>GetPetName</code>	83
Добавление тестовых записей	84
Модель фабрики поставщиков данных ADO.NET	85
Полный пример фабрики поставщиков данных	86
Потенциальный недостаток модели фабрики поставщиков данных	90
Погружение в детали объектов подключений, команд и чтения данных	91
Работа с объектами подключений	92
Работа с объектами команд	95
Работа с объектами чтения данных	96
Работа с запросами создания, обновления и удаления	99
Создание классов <code>Car</code> и <code>CarViewModel</code>	99
Добавление класса <code>InventoryDal</code>	100
Добавление реализации <code>IDisposable</code>	101
Создание строго типизированного метода <code>InsertCar()</code>	104
Добавление логики удаления	104
Добавление логики обновления	105
Работа с параметризованными объектами команд	106
Выполнение хранимой процедуры	109
Создание консольного клиентского приложения	111
Понятие транзакций базы данных	112
Основные члены объекта транзакции ADO.NET	113
Добавление метода транзакции в <code>InventoryDal</code>	113
Тестирование транзакции базы данных	116
Выполнение массового копирования с помощью ADO.NET	117
Исследование класса <code>SqlBulkCopy</code>	117
Создание специального класса чтения данных	117

8 Содержание

Выполнение массового копирования	121
Тестирование массового копирования	122
Резюме	123
Часть VII. Entity Framework Core	125
Глава 22. Введение в Entity Framework Core	126
Инструменты объектно-реляционного отображения	127
Роль Entity Framework Core	128
Строительные блоки Entity Framework Core	129
Класс DbContext	129
Поддержка транзакций и точек сохранения	133
Транзакции и стратегии выполнения	133
Класс DbSet<T>	134
Экземпляр ChangeTracker	137
Сущности	137
Выполнение запросов	158
Смешанное выполнение на клиентской и серверной сторонах	159
Сравнение отслеживаемых и неотслеживаемых запросов	159
Важные функциональные средства EF Core	160
Обработка значений, генерируемых базой данных	160
Проверка параллелизма	161
Устойчивость подключений	162
Связанные данные	163
Глобальные фильтры запросов	166
Выполнение низкоуровневых запросов SQL с помощью LINQ	168
Пакетирование операторов	169
Принадлежащие сущностные типы	170
Сопоставление с функциями базы данных	173
Команды CLI глобального инструмента EF Core	173
Команды для управления миграциями	175
Команды для управления базой данных	179
Команды для управления типами DbContext	180
Резюме	182
Глава 23. Построение уровня доступа к данным с помощью Entity Framework Core	183
“Сначала код” или “сначала база данных”	183
Создание проектов AutoLot.Dal и AutoLot.Models	184
Создание шаблонов для класса, производного от DbContext, и сущностных классов	185
Переключение на подход “сначала код”	186
Создание фабрики экземпляров класса, производного от DbContext, на этапе проектирования	186

Создание начальной миграции	187
Применение миграции	187
Обновление модели	188
Сущности	188
Класс ApplicationDbContext	197
Создание миграции и обновление базы данных	203
Добавление представления базы данных и хранимой процедуры	204
Добавление класса MigrationHelpers	204
Обновление и применение миграции	205
Добавление модели представления	206
Добавление класса модели представления	206
Добавление класса модели представления к ApplicationDbContext	207
Добавление хранилищ	207
Добавление базового интерфейса IRepo	208
Добавление класса BaseRepo	208
Интерфейсы хранилищ, специфичных для сущностей	212
Реализация классов хранилищ, специфичных для сущностей	213
Программная работа с базой данных и миграциями	218
Удаление, создание и очистка базы данных	219
Инициализация базы данных	220
Создание выборочных данных	220
Загрузка выборочных данных	221
Настройка тестов	223
Создание проекта	223
Конфигурирование проекта	224
Создание класса TestHelpers	224
Добавление класса BaseTest	226
Добавление класса тестовой оснастки EnsureAutoLotDatabase	227
Добавление классов интеграционных тестов	228
Выполнение тестов	231
Запрашивание базы данных	231
Состояние сущности	232
Запросы LINQ	232
Выполнение запросов SQL с помощью LINQ	250
Методы агрегирования	251
Any () и All ()	253
Получение данных из хранимых процедур	254
Создание записей	255
Состояние сущности	255
Добавление одной записи	255
Добавление одной записи с использованием метода Attach ()	256
Добавление нескольких записей одновременно	257
Соображения относительно столбца идентичности при добавлении записей	258
Добавление объектного графа	258

10 Содержание

Обновление записей	259
Состояние сущности	259
Обновление отслеживаемых сущностей	259
Обновление неотслеживаемых сущностей	260
Проверка параллелизма	261
Удаление записей	262
Состояние сущности	262
Удаление отслеживаемых сущностей	263
Удаление неотслеживаемых сущностей	263
Перехват отказов каскадного удаления	264
Проверка параллелизма	264
Резюме	264
Часть VIII. Разработка клиентских приложений для Windows	265
Глава 24. Введение в Windows Presentation Foundation и XAML	266
Побудительные причины создания WPF	266
Унификация несходных API-интерфейсов	267
Обеспечение разделения обязанностей через XAML	268
Обеспечение оптимизированной модели визуализации	268
Упрощение программирования сложных пользовательских интерфейсов	269
Исследование сборок WPF	270
Роль класса Application	272
Построение класса приложения	272
Перечисление элементов коллекции Windows	273
Роль класса Window	273
Синтаксис XAML для WPF	278
Введение в XAML	278
Пространства имен XML и "ключевые слова" XAML	280
Управление видимостью классов и переменных-членов	282
Элементы XAML, атрибуты XAML и преобразователи типов	283
Понятие синтаксиса "свойство-элемент" в XAML	284
Понятие присоединяемых свойств XAML	285
Понятие расширений разметки XAML	286
Построение приложений WPF с использованием Visual Studio	288
Шаблоны проектов WPF	288
Панель инструментов и визуальный конструктор/редактор XAML	288
Установка свойств с использованием окна Properties	289
Обработка событий с использованием окна Properties	290
Обработка событий в редакторе XAML	291
Окно Document Outline	292
Включение и отключение отладчика XAML	292
Исследование файла App.xaml	293
Отображение разметки XAML окна на код C#	294
Роль BAML	296

Разгадывание загадки Main ()	297
Взаимодействие с данными уровня приложения	297
Обработка закрытия объекта Window	298
Перехват событий мыши	299
Перехват событий клавиатуры	300
Резюме	301
Глава 25. Элементы управления, компоновки, события и привязка данных в WPF	302
Обзор основных элементов управления WPF	302
Элементы управления для работы с Ink API	303
Элементы управления для работы с документами WPF	304
Общие диалоговые окна WPF	304
Краткий обзор визуального конструктора WPF в Visual Studio	304
Работа с элементами управления WPF в Visual Studio	305
Работа с окном Document Outline	306
Управление компоновкой содержимого с использованием панелей	306
Позиционирование содержимого внутри панелей Canvas	308
Позиционирование содержимого внутри панелей WrapPanel	310
Позиционирование содержимого внутри панелей StackPanel	311
Позиционирование содержимого внутри панелей Grid	312
Панели Grid с типами GridSplitter	314
Позиционирование содержимого внутри панелей DockPanel	315
Включение прокрутки в типах панелей	316
Конфигурирование панелей с использованием визуальных конструкторов Visual Studio	317
Построение окна с использованием вложенных панелей	319
Построение системы меню	320
Визуальное построение меню	322
Построение панели инструментов	322
Построение строки состояния	323
Завершение проектирования пользовательского интерфейса	323
Реализация обработчиков событий MouseEnter/MouseLeave	324
Реализация логики проверки правописания	324
Понятие команд WPF	325
Внутренние объекты команд	326
Подключение команд к свойству Command	327
Подключение команд к произвольным действиям	327
Работа с командами Open и Save	328
Понятие маршрутизируемых событий	330
Роль пузырьковых маршрутизируемых событий	332
Продолжение или прекращение пузырькового распространения	333
Роль туннельных маршрутизируемых событий	333
Более глубокое исследование	
API-интерфейсов и элементов управления WPF	335
Работа с элементом управления TabControl	335

12 Содержание

Построение вкладки Ink API	336
Проектирование панели инструментов	336
Элемент управления RadioButton	337
Добавление кнопок сохранения, загрузки и удаления	337
Добавление элемента управления InkCanvas	338
Предварительный просмотр окна	338
Обработка событий для вкладки Ink API	338
Добавление элементов управления в панель инструментов	339
Элемент управления InkCanvas	340
Элемент управления ComboBox	341
Сохранение, загрузка и очистка данных InkCanvas	343
Введение в модель привязки данных WPF	344
Построение вкладки Data Binding	345
Установка привязки данных	345
Свойство DataContext	346
Форматирование привязанных данных	347
Преобразование данных с использованием интерфейса IValueConverter	347
Установление привязок данных в коде	349
Построение вкладки DataGrid	350
Роль свойств зависимости	352
Исследование существующего свойства зависимости	354
Важные замечания относительно оболочек свойств CLR	357
Построение специального свойства зависимости	357
Добавление процедуры проверки достоверности данных	360
Реагирование на изменение свойства	361
Резюме	362
Глава 26. Службы визуализации графики WPF	363
Понятие служб визуализации графики WPF	363
Варианты графической визуализации WPF	364
Визуализация графических данных с использованием фигур	365
Добавление прямоугольников, эллипсов и линий на поверхность Canvas	366
Удаление прямоугольников, эллипсов и линий с поверхности Canvas	370
Работа с элементами Polyline и Polygon	371
Работа с элементом Path	371
Кисти и перья WPF	375
Конфигурирование кистей с использованием Visual Studio	375
Конфигурирование кистей в коде	377
Конфигурирование перьев	378
Применение графических трансформаций	378
Первый взгляд на трансформации	379
Трансформация данных Canvas	380
Работа с редактором трансформаций Visual Studio	382
Построение начальной компоновки	382
Применение трансформаций на этапе проектирования	384
Трансформация холста в коде	385

Визуализация графических данных с использованием рисунков и геометрических объектов	386
Построение кисти DrawingBrush с использованием геометрических объектов	387
Рисование с помощью DrawingBrush	388
Включение типов Drawing в DrawingImage	388
Работа с векторными изображениями	389
Преобразование файла с векторной графикой в файл XAML	389
Импортирование графических данных в проект WPF	390
Взаимодействие с изображением	391
Визуализация графических данных с использованием визуального уровня	391
Базовый класс Visual и производные дочерние классы	392
Первый взгляд на класс DrawingVisual	392
Визуализация графических данных в специальном диспетчере компоновки	394
Реагирование на операции проверки попадания	397
Резюме	398
Глава 27. Ресурсы, анимация, стили и шаблоны WPF	399
Система ресурсов WPF	399
Работа с двоичными ресурсами	400
Работа с объектными (логическими) ресурсами	404
Роль свойства Resources	404
Определение ресурсов уровня окна	405
Расширение разметки {StaticResource}	407
Расширение разметки {DynamicResource}	408
Ресурсы уровня приложения	408
Определение объединенных словарей ресурсов	409
Определение сборки, включающей только ресурсы	410
Службы анимации WPF	411
Роль классов анимации	412
Свойства To, From и By	412
Роль базового класса Timeline	413
Реализация анимации в коде C#	413
Управление темпом анимации	415
Запуск в обратном порядке и циклическое выполнение анимации	416
Реализация анимации в разметке XAML	416
Роль раскадровок	417
Роль триггеров событий	418
Анимация с использованием дискретных ключевых кадров	418
Роль стилей WPF	419
Определение и применение стиля	420
Переопределение настроек стиля	421
Влияние атрибута TargetType на стили	421
Создание подклассов существующих стилей	422

14 Содержание

Определение стилей с триггерами	423
Определение стилей с множеством триггеров	424
Стили с анимацией	424
Применение стилей в коде	425
Логические деревья, визуальные деревья и стандартные шаблоны	426
Программное инспектирование логического дерева	427
Программное инспектирование визуального дерева	429
Программное инспектирование стандартного шаблона элемента управления	429
Построение шаблона элемента управления с помощью инфраструктуры триггеров	432
Шаблоны как ресурсы	433
Встраивание визуальных подсказок с использованием триггеров	434
Роль расширения разметки {TemplateBinding}	435
Роль класса ContentPresenter	436
Встраивание шаблонов в стили	436
Резюме	437
Глава 28. Уведомления WPF, проверка достоверности, команды и MVVM	439
Введение в паттерн MVVM	439
Модель	440
Представление	440
Модель представления	440
Анемичные модели или анемичные модели представлений	440
Система уведомлений привязки WPF	441
Наблюдаемые модели и коллекции	441
Добавление привязок и данных	443
Изменение данных об автомобиле в коде	444
Наблюдаемые модели	445
Наблюдаемые коллекции	447
Итоговые сведения об уведомлениях и наблюдаемых моделях	449
Проверка достоверности WPF	450
Модификация примера для демонстрации проверки достоверности	450
Класс Validation	450
Варианты проверки достоверности	451
Использование аннотаций данных в WPF	461
Настройка свойства ErrorTemplate	463
Итоговые сведения о проверке достоверности	465
Создание специальных команд	465
Реализация интерфейса ICommand	465
Добавление класса ChangeColorCommand	466
Создание класса CommandBase	468
Добавление класса AddCarCommand	469

Объекты RelayCommand	470
Итоговые сведения о командах	472
Перенос кода и данных в модель представления	472
Перенос кода MainWindow.xaml.cs	473
Обновление кода и разметки MainWindow	473
Обновление разметки элементов управления	474
Итоговые сведения о моделях представлений	474
Обновление проекта AutoLot.Dal для MVVM	475
Резюме	475
Часть IX. ASP.NET Core	477
Глава 29. Введение в ASP.NET Core	478
Краткий экскурс в прошлое	478
Введение в паттерн MVC	478
ASP.NET Core и паттерн MVC	479
ASP.NET Core и .NET Core	479
Одна инфраструктура, много сценариев использования	480
Функциональные средства ASP.NET Core из MVC/Web API	480
Соглашения по конфигурации	481
Привязка моделей	484
Проверка достоверности моделей	488
Маршрутизация	489
Фильтры	495
Нововведения в ASP.NET Core	496
Встроенное внедрение зависимостей	496
Осведомленность о среде	497
Конфигурация приложений	499
Развертывание приложений ASP.NET Core	500
Легковесный и модульный конвейер запросов HTTP	500
Создание и конфигурирование решения	500
Использование Visual Studio	500
Использование командной строки	503
Запуск приложений ASP.NET Core	504
Конфигурирование настроек запуска	505
Использование Visual Studio	505
Использование командной строки	506
или окна терминала Visual Studio Code	506
Использование Visual Studio Code	506
Отладка приложений ASP.NET Core	507
Обновление портов AutoLot.Api	508
Создание и конфигурирование экземпляра WebHost	508
Файл Program.cs	508
Файл Startup.cs	509
Ведение журнала	516
Резюме	529

Глава 30. Создание служб REST с помощью ASP.NET Core	530
Введение в REST-службы ASP.NET Core	530
Создание действий контроллера с использованием служб REST	531
Результаты ответов в формате JSON	531
Атрибут ApiController	533
Обновление настроек Swagger/OpenAPI	536
Обновление обращений к Swagger в классе Startup	536
Добавление файла XML-документации	537
Добавление XML-комментариев в процесс генерации Swagger	539
Дополнительные возможности документирования для конечных точек API	540
Построение методов действий API	542
Конструктор	543
Методы GetXXX ()	543
Метод UpdateOne ()	544
Метод AddOne ()	545
Метод DeleteOne ()	546
Класс CarsController	547
Оставшиеся контроллеры	548
Фильтры исключений	550
Создание специального фильтра исключений	551
Тестирование фильтра исключений	553
Добавление поддержки запросов между источниками	553
Создание политики CORS	553
Добавление политики CORS в конвейер обработки HTTP	554
Резюме	554
Глава 31. Создание приложений MVC с помощью ASP.NET Core	555
Введение в представления ASP.NET Core	555
Экземпляры класса ViewResult и методы действий	555
Механизм визуализации и синтаксис Razor	558
Представления	561
Компоновки	565
Частичные представления	566
Обновление компоновки с использованием частичных представлений	567
Отправка данных представлениям	568
Вспомогательные функции дескрипторов	570
Включение вспомогательных функций дескрипторов	570
Вспомогательная функция дескриптора для формы	576
Вспомогательная функция дескриптора для действия формы	577
Вспомогательная функция дескриптора для якоря	578
Вспомогательная функция дескриптора для элемента ввода	578
Вспомогательная функция дескриптора для текстовой области	579
Вспомогательная функция дескриптора для элемента выбора	579
Вспомогательные функции дескрипторов для проверки достоверности	580
Вспомогательная функция дескриптора для среды	581

Вспомогательная функция дескриптора для ссылки	582
Вспомогательная функция дескриптора для сценария	582
Вспомогательная функция дескриптора для изображения	584
Специальные вспомогательные функции дескрипторов	584
Подготовительные шаги	584
Создание базового класса	585
Вспомогательная функция дескриптора для вывода сведений об элементе	586
Вспомогательная функция дескриптора для удаления элемента	587
Вспомогательная функция дескриптора для редактирования сведений об элементе	588
Вспомогательная функция дескриптора для создания элемента	588
Вспомогательная функция дескриптора для вывода списка элементов	589
Обеспечение видимости специальных вспомогательных функций дескрипторов	590
Вспомогательные функции HTML	590
Вспомогательная функция <code>DisplayFor()</code>	591
Вспомогательная функция <code>DisplayForModel()</code>	591
Вспомогательные функции <code>EditorFor()</code> и <code>EditorForModel()</code>	591
Управление библиотеками клиентской стороны	591
Установка диспетчера библиотек как глобального инструмента .NET Core	592
Добавление в проект AutoLot.Mvc библиотек клиентской стороны	592
Завершение работы над представлениями CarsController и Cars	596
Класс <code>CarsController</code>	596
Частичное представление списка автомобилей	597
Представление <code>Index</code>	598
Представление <code>ByMake</code>	599
Представление <code>Details</code>	600
Представление <code>Create</code>	602
Представление <code>Edit</code>	604
Представление <code>Delete</code>	606
Компоненты представлений	607
Код серверной стороны	608
Построение частичного представления	609
Вызов компонентов представлений	610
Вызов компонентов представлений как специальных вспомогательных функций дескрипторов	610
Обновление меню	610
Пакетирование и минификация	611
Пакетирование	611
Минификация	611
Решение <code>WebOptimizer</code>	611
Шаблон параметров в ASP.NET Core	613
Добавление информации об автодилере	613
Создание оболочки службы	615
Обновление конфигурации приложения	615
Создание класса <code>ApiServiceSettings</code>	616

18 Содержание

Оболочка службы API	616
Конфигурирование служб	620
Построение класса CarsController	621
Вспомогательный метод GetMakes()	622
Вспомогательный метод GetOneCar()	622
Открытые методы действий	622
Обновление компонента представления	624
Совместный запуск приложений AutoLot.Mvc и AutoLot.Api	625
Использование Visual Studio	625
Использование командной строки	626
Резюме	626
Предметный указатель	627

ЧАСТЬ VI

Работа с файлами, сериализация объектов и доступ к данным

ГЛАВА 20

Файловый ввод-вывод и сериализация объектов

При создании настольных приложений возможность сохранения информации между пользовательскими сеансами является привычным делом. В настоящей главе рассматривается несколько тем, касающихся ввода-вывода, с точки зрения платформы .NET Core. Первая задача связана с исследованием основных типов, определенных в пространстве имен `System.IO`, с помощью которых можно программно модифицировать структуру каталогов и файлов. Вторая задача предусматривает изучение разнообразных способов чтения и записи символьных, двоичных, строковых и находящихся в памяти структур данных.

После изучения способов манипулирования файлами и каталогами с использованием основных типов ввода-вывода вы ознакомитесь со связанной темой — *сериализацией объектов*. Сериализацию объектов можно применять для сохранения и извлечения состояния объекта с помощью любого типа, производного от `System.IO.Stream`.

На заметку! Чтобы можно было успешно выполнять примеры в главе, IDE-среда Visual Studio должна быть запущена с правами администратора (для этого нужно просто щелкнуть правой кнопкой мыши на значке Visual Studio и выбрать в контекстном меню пункт Запуск от имени администратора). В противном случае при доступе к файловой системе компьютера могут возникать исключения, связанные с безопасностью.

Исследование пространства имен `System.IO`

В рамках платформы .NET Core пространство имен `System.IO` представляет собой раздел библиотек базовых классов, выделенный службам файлового ввода и вывода, а также ввода и вывода в памяти. Подобно любому пространству имен внутри `System.IO` определен набор классов, интерфейсов, перечислений, структур и дегиков, большинство из которых находятся в сборке `mscorlib.dll`. В дополнение к типам, содержащимся внутри `mscorlib.dll`, в сборке `System.dll` определены дополнительные члены пространства имен `System.IO`.

Многие типы из пространства имен `System.IO` сосредоточены на программной манипуляции физическими каталогами и файлами. Тем не менее, дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также в области памяти. В табл. 20.1 кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности, доступной в пространстве имен `System.IO`.

Таблица 20.1. Основные члены пространства имен System.IO

Неабстрактные классы ввода-вывода	Описание
BinaryReader BinaryWriter	Эти классы позволяют сохранять и извлекать данные элементарных типов (целочисленные, булевские, строковые и т.д.) как двоичные значения
BufferedStream	Этот класс предоставляет временное хранилище для потока байтов, который может быть зафиксирован в постоянном хранилище в более позднее время
Directory DirectoryInfo	Эти классы применяются для манипулирования структурой каталогов машины. Тип Directory открывает функциональность с использованием <i>статических членов</i> . Тип DirectoryInfo обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
DriveInfo	Этот класс предоставляет детальную информацию о дисковых устройствах, присутствующих на заданной машине
File FileInfo	Эти классы служат для манипулирования набором файлов на машине. Тип File открывает функциональность через <i>статические члены</i> . Тип FileInfo обеспечивает аналогичную функциональность через действительную <i>объектную ссылку</i>
FileStream	Этот класс предоставляет произвольный доступ к файлу (например, с возможностями поиска) с данными, представленными в виде потока байтов
FileSystemWatcher	Этот класс позволяет отслеживать модификацию внешних файлов в указанном каталоге
MemoryStream	Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле
Path	Этот класс выполняет операции над типами System.String, которые содержат информацию о пути к файлу или каталогу, в независимой от платформы манере
StreamWriter StreamReader	Эти классы применяются для хранения (и извлечения) текстовой информации в файле. Они не поддерживают произвольный доступ к файлу
StringWriter StringReader	Подобно StreamWriter/StreamReader эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл

В дополнение к описанным конкретным классам внутри System.IO определено несколько перечислений, а также набор абстрактных классов (скажем, Stream, TextReader и TextWriter), которые формируют разделяемый полиморфный интерфейс для всех наследников. В главе вы узнаете о многих типах пространства имен System.IO.

Классы Directory (DirectoryInfo) и File (FileInfo)

Пространство имен System.IO предлагает четыре класса, которые позволяют манипулировать индивидуальными файлами, а также взаимодействовать со структурой каталогов машины. Первые два класса, Directory и File, открывают доступ к операциям создания, удаления, копирования и перемещения через разнообразные статические члены. Тесно связанные с ними классы FileInfo и DirectoryInfo обеспечивают похожую функциональность в виде методов уровня экземпляра (следовательно, их экземпляры придется создавать с помощью ключевого слова new). Классы Directory и File непосредственно расширяют класс System.Object, в то время как DirectoryInfo и FileInfo являются производными от абстрактного класса FileSystemInfo.

Обычно классы FileInfo и DirectoryInfo считаются лучшим выбором для получения полных сведений о файле или каталоге (например, времени создания или возможности чтения/записи), т.к. их члены возвращают строго типизированные объекты. В отличие от них члены классов Directory и File, как правило, возвращают простые строковые значения, а не строго типизированные объекты. Тем не менее, это всего лишь рекомендация; во многих случаях одну и ту же работу можно делать с использованием File/FileInfo или Directory/DirectoryInfo.

Абстрактный базовый класс FileSystemInfo

Классы DirectoryInfo и FileInfo получают многие линии поведения от абстрактного базового класса FileSystemInfo. По большей части члены класса FileSystemInfo применяются для выяснения общих характеристик (таких как время создания, разнообразные атрибуты и т.д.) заданного файла или каталога. В табл. 20.2 перечислены некоторые основные свойства, представляющие интерес.

Таблица 20.2. Избранные свойства класса FileSystemInfo

Свойство	Описание
Attributes	Получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением FileMode (например, доступный только для чтения, зашифрованный, скрытый или сжатый)
CreationTime	Получает или устанавливает время создания текущего файла или каталога
Exists	Определяет, существует ли данный файл или каталог
Extension	Извлекает расширение файла
FullName	Получает полный путь к файлу или каталогу
LastAccessTime	Получает или устанавливает время последнего доступа к текущему файлу или каталогу
LastWriteTime	Получает или устанавливает время последней записи в текущий файл или каталог
Name	Получает имя текущего файла или каталога

В классе `FileSystemInfo` также определен метод `Delete()`. Он реализуется производными типами для удаления заданного файла или каталога с жесткого диска. Кроме того, перед получением информации об атрибутах можно вызвать метод `Refresh()`, чтобы обеспечить актуальность статистических данных о текущем файле или каталоге.

Работа с типом `DirectoryInfo`

Первый неабстрактный тип, связанный с вводом-выводом, который мы исследуем здесь — `DirectoryInfo`. Этот класс содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. В дополнение к функциональности, предоставленной его базовым классом (`FileSystemInfo`), класс `DirectoryInfo` предлагает ключевые члены, описанные в табл. 20.3.

Таблица 20.3. Основные члены типа `DirectoryInfo`

Член	Описание
<code>Create()</code>	Создает каталог (или набор подкаталогов) по заданному путевому имени
<code>CreateSubdirectory()</code>	
<code>Delete()</code>	Удаляет каталог и все его содержимое
<code>GetDirectories()</code>	Возвращает массив объектов <code>DirectoryInfo</code> , которые представляют все подкаталоги в текущем каталоге
<code>GetFiles()</code>	Извлекает массив объектов <code>FileInfo</code> , представляющий набор файлов в заданном каталоге
<code>MoveTo()</code>	Перемещает каталог со всем содержимым в новый путь
<code>Parent</code>	Извлекает родительский каталог данного каталога
<code>Root</code>	Получает корневую часть пути

Работа с типом `DirectoryInfo` начинается с указания отдельного пути в параметре конструктора. Если требуется получить доступ к текущему рабочему каталогу (каталогу выполняющегося приложения), то следует применять обозначение в виде точки (.). Вот некоторые примеры:

```
// Привязаться к текущему рабочему каталогу.
DirectoryInfo dir1 = new DirectoryInfo(".");
// Привязаться к C:\Windows, используя дословную строку.
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

Во втором примере предполагается, что путь, передаваемый конструктору (`C:\Windows`), уже существует на физической машине. Однако при попытке взаимодействия с несуществующим каталогом генерируется исключение `System.IO.DirectoryNotFoundException`. Таким образом, чтобы указать каталог, который пока еще не создан, перед работой с ним понадобится вызвать метод `Create()`:

```
// Привязаться к несуществующему каталогу, затем создать его.
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

24 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

Синтаксис пути, используемый в предыдущем примере, ориентирован на Windows. Если вы разрабатываете приложения .NET Core для разных платформ, тогда должны применять конструкции Path.VolumeSeparatorChar и Path.DirectorySeparatorChar, которые будут выдавать подходящие символы на основе платформы. Модифицируйте предыдущий код, как показано ниже:

```
DirectoryInfo dir3 = new DirectoryInfo(  
    $@"C{Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}  
MyCode{Path.DirectorySeparatorChar}Testing");
```

После создания объекта DirectoryInfo можно исследовать содержимое лежащего в основе каталога с помощью любого свойства, унаследованного от FileInfo. В целях иллюстрации создайте новый проект консольного приложения по имени DirectoryApp и импортируйте в файл кода C# пространства имен System и System.IO. Измените класс Program, добавив представленный далее новый статический метод, который создает объект DirectoryInfo, отображенный на C:\Windows (при необходимости подкорректируйте путь), и выводит интересные статистические данные:

```
using System;  
using System.IO;  
  
Console.WriteLine("***** Fun with Directory(Info) *****\n");  
ShowWindows DirectoryInfo();  
Console.ReadLine();  
  
static void ShowWindows DirectoryInfo()  
{  
    // Вывести информацию о каталоге. В случае работы не под  
    // управлением Windows подключитесь к другому каталогу.  
    DirectoryInfo dir = new DirectoryInfo($@"C{Path.VolumeSeparatorChar}  
{Path.DirectorySeparatorChar}Windows");  
    Console.WriteLine("***** Directory Info *****");  
        // Информация о каталоге  
    Console.WriteLine("FullName: {0}", dir.FullName);      // Полное имя  
    Console.WriteLine("Name: {0}", dir.Name);           // Имя каталога  
    Console.WriteLine("Parent: {0}", dir.Parent);  
        // Родительский каталог  
    Console.WriteLine("Creation: {0}", dir.CreationTime);  
        // Время создания  
    Console.WriteLine("Attributes: {0}", dir.Attributes); // Атрибуты  
    Console.WriteLine("Root: {0}", dir.Root);           // Корневой каталог  
    Console.WriteLine("*****\n");  
}
```

Вывод у вас может отличаться, но быть похожим:

```
***** Fun with Directory(Info) *****  
***** Directory Info *****  
FullName: C:\Windows  
Name: Windows  
Parent:  
Creation: 3/19/2019 00:37:22  
Attributes: Directory  
Root: C:\  
*****
```

Перечисление файлов с помощью типа DirectoryInfo

В дополнение к получению базовых сведений о существующем каталоге текущий пример можно расширить, чтобы задействовать некоторые методы типа DirectoryInfo. Первым делом мы используем метод GetFiles() для получения информации обо всех файлах *.jpg, расположенных в каталоге C:\Windows\Web\Wallpaper.

На заметку! Если вы не работаете на машине с Windows, тогда модифицируйте код, чтобы читать файлы в каком-нибудь каталоге на вашей машине. Не забудьте использовать Path.VolumeSeparatorChar и Path.DirectorySeparatorChar, сделав код межплатформенным.

Метод GetFiles() возвращает массив объектов FileInfo, каждый из которых открывает доступ к детальной информации о конкретном файле (тип FileInfo будет подробно описан далее в главе). Создайте в классе Program следующий статический метод:

```
static void DisplayImageFiles()
{
    DirectoryInfo dir = new
        DirectoryInfo(@"C:\Windows\Web\Wallpaper");
    // Получить все файлы с расширением *.jpg.
    FileInfo[] imageFiles =
        dir.GetFiles("*.jpg", SearchOption.AllDirectories);
    // Сколько файлов найдено?
    Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);
    // Вывести информацию о каждом файле.
    foreach (FileInfo f in imageFiles)
    {
        Console.WriteLine("*****");
        Console.WriteLine("File name: {0}", f.Name);           // Имя файла
        Console.WriteLine("File size: {0}", f.Length);        // Размер
        Console.WriteLine("Creation: {0}", f.CreationTime);   // Время создания
        Console.WriteLine("Attributes: {0}", f.Attributes);   // Атрибуты
        Console.WriteLine("*****\n");
    }
}
```

Обратите внимание на указание в вызове GetFiles() варианта поиска; SearchOption.AllDirectories обеспечивает просмотр всех подкаталогов корня. В результате запуска приложения выводится список файлов, которые соответствуют поисковому шаблону.

Создание подкаталогов с помощью типа DirectoryInfo

Посредством метода DirectoryInfo.CreateSubdirectory() можно программно расширять структуру каталогов. Он позволяет создавать одиничный подкаталог, а также множество вложенных подкаталогов в единственном вызове. В приведенном ниже методе демонстрируется расширение структуры каталога, в котором запускается приложение (обозначаемого с помощью .), несколькими специальными подкаталогами:

```
static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(".");
    // Создать \MyFolder в каталоге запуска приложения.
    dir.CreateSubdirectory("MyFolder");
    // Создать \MyFolder2\Data в каталоге запуска приложения.
    dir.CreateSubdirectory(
        $"{Path.DirectorySeparatorChar}Data");
}
```

Получать возвращаемое значение метода `CreateSubdirectory()` не обязательно, но важно знать, что в случае его успешного выполнения возвращается объект `DirectoryInfo`, представляющий вновь созданный элемент. Взгляните на следующую модификацию предыдущего метода:

```
static void ModifyAppDirectory()
{
    DirectoryInfo dir = new DirectoryInfo(".");
    // Создать \MyFolder в начальном каталоге.
    dir.CreateSubdirectory("MyFolder");
    // Получите возвращенный объект DirectoryInfo.
    DirectoryInfo myDataFolder = dir.CreateSubdirectory(
        $"{Path.DirectorySeparatorChar}Data");
    // Выводит путь к ..\MyFolder2\Data.
    Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

Вызвав метод `ModifyAppDirectory()` в операторах верхнего уровня и запустив программу, в проводнике Windows можно будет увидеть новые подкаталоги.

Работа с типом `Directory`

Вы видели тип `DirectoryInfo` в действии и теперь готовы к изучению типа `Directory`. По большей части статические члены типа `Directory` воспроизводят функциональность, которая предоставляется членами уровня экземпляра, определенными в `DirectoryInfo`. Тем не менее, вспомните, что члены типа `Directory` обычно возвращают строковые данные, а не строго типизированные объекты `FileInfo`/`DirectoryInfo`.

Давайте взглянем на функциональность типа `Directory`: показанный ниже вспомогательный метод отображает имена всех логических устройств на текущем компьютере (с помощью метода `Directory.GetLogicalDrives()`) и применяет статический метод `Directory.Delete()` для удаления созданных ранее подкаталогов `\MyFolder` и `\MyFolder2\Data`:

```
static void FunWithDirectoryType()
{
    // Вывести список всех логических устройств на текущем компьютере.
    string[] drives = Directory.GetLogicalDrives();
    Console.WriteLine("Here are your drives:");
    foreach (string s in drives)
    {
        Console.WriteLine("--> {0} ", s);
    }
}
```

```
// Удалить ранее созданные подкаталоги.
Console.WriteLine("Press Enter to delete directories");
Console.ReadLine();
try
{
    Directory.Delete("MyFolder");
    // Второй параметр указывает, нужно ли удалять внутренние подкаталоги
    Directory.Delete("MyFolder2", true);
}
catch (IOException e)
{
    Console.WriteLine(e.Message);
}
```

Работа с типом DriveInfo

Пространство имен System.IO содержит класс по имени DriveInfo. Подобно Directory.GetLogicalDrives() статический метод DriveInfo.GetDrives() позволяет выяснить имена устройств на машине. Однако в отличие от Directory.GetLogicalDrives() метод DriveInfo.GetDrives() предоставляет множество дополнительных деталей (например, тип устройства, доступное свободное пространство и метка тома). Взгляните на следующие операторы верхнего уровня в новом проекте консольного приложения DriveInfoApp:

```
using System;
using System.IO;
// Получить информацию обо всех устройствах.
DriveInfo[] myDrives = DriveInfo.GetDrives();
// Вывести сведения об устройствах.
foreach(DriveInfo d in myDrives)
{
    Console.WriteLine("Name: {0}", d.Name);           // имя
    Console.WriteLine("Type: {0}", d.DriveType);      // тип
    // Проверить, смонтировано ли устройство.
    if(d.IsReady)
    {
        Console.WriteLine("Free space: {0}", d.TotalFreeSpace); // свободное пространство
        Console.WriteLine("Format: {0}", d.DriveFormat);       // формат устройства
        Console.WriteLine("Label: {0}", d.VolumeLabel);        // метка тома
    }
    Console.WriteLine();
}
```

Вот возможный вывод:

```
Name: C:\  
Type: Fixed  
Free space: 284131119104  
Format: NTFS  
Label: OS
```

```
Name: M:\  
Type: Network  
Free space: 4711871942656  
Format: NTFS  
Label: DigitalMedia
```

К этому моменту вы изучили несколько основных линий поведения классов `Directory`, `DirectoryInfo` и `DriveInfo`. Далее вы ознакомитесь с тем, как создавать, открывать, закрывать и удалять файлы, находящиеся в заданном каталоге.

Работа с типом `FileInfo`

Как было показано в предыдущем примере `DirectoryApp`, класс `FileInfo` позволяет получать сведения о существующих файлах на жестком диске (такие как время создания, размер и атрибуты) и помогает создавать, копировать, перемещать и удалять файлы. В дополнение к набору функциональности, унаследованной от `FileSystemInfo`, класс `FileInfo` имеет ряд уникальных членов, которые описаны в табл. 20.4.

Таблица 20.4. Основные члены `FileInfo`

Член	Описание
<code>AppendText ()</code>	Создает объект <code>StreamWriter</code> (описанный далее в главе) и добавляет текст в файл
<code>CopyTo ()</code>	Копирует существующий файл в новый файл
<code>Create ()</code>	Создает новый файл и возвращает объект <code>FileStream</code> (описанный далее в главе) для взаимодействия с вновь созданным файлом
<code>CreateText ()</code>	Создает объект <code>StreamWriter</code> , который производит запись в новый текстовый файл
<code>Delete ()</code>	Удаляет файл, к которому привязан экземпляр <code>FileInfo</code>
<code>Directory</code>	Получает экземпляр родительского каталога
<code>DirectoryName</code>	Получает полный путь к родительскому каталогу
<code>Length</code>	Получает размер текущего файла
<code>MoveTo ()</code>	Перемещает указанный файл в новое местоположение, предоставляемая возможность указания нового имени для файла
<code>Name</code>	Получает имя файла
<code>Open ()</code>	Открывает файл с разнообразными привилегиями чтения/записи и совместного доступа
<code>OpenRead ()</code>	Создает объект <code>FileStream</code> , доступный только для чтения
<code>OpenText ()</code>	Создает объект <code>StreamReader</code> (описанный далее в главе), который производит чтение из существующего текстового файла
<code>OpenWrite ()</code>	Создает объект <code>FileStream</code> , доступный только для записи

Обратите внимание, что большинство методов класса `FileInfo` возвращают специфический объект ввода-вывода (например, `FileStream` и `StreamWriter`), который позволяет начать чтение и запись данных в ассоциированный файл во множестве форматов. Вскоре мы исследуем указанные типы, но прежде чем рассмотреть

работающий пример, давайте изучим различные способы получения дескриптора файла с использованием класса `FileInfo`.

Метод `FileInfo.Create()`

Следующий набор примеров находится в проекте консольного приложения по имени `SimpleFileIO`. Один из способов создания дескриптора файла предусматривает применение метода `FileInfo.Create()`:

```
using System;
using System.IO;
Console.WriteLine("***** Simple IO with the File Type *****\n");

// Измените это на папку на своей машине, к которой вы имеете доступ
// по чтению/записи или запускайте приложение от имени администратора.
var fileName = $"{Path.VolumeSeparatorChar}
{Path.DirectorySeparatorChar}temp{Path.DirectorySeparatorChar}Test.dat";

// Создать новый файл на диске C: .
FileInfo f = new FileInfo(fileName);
FileStream fs = f.Create();
// Использовать объект FileStream...
// Закрыть файловый поток.
fs.Close();
```

На заметку! В зависимости от имеющихся у вас пользовательских разрешений и конфигурации системы примеры, которые здесь рассматриваются, могут требовать запуска Visual Studio от имени администратора.

Метод `FileInfo.Create()` возвращает тип `FileStream`, который предоставляет синхронную и асинхронную операции записи/чтения лежащего в его основе файла. Имейте в виду, что объект `FileStream`, возвращаемый `FileInfo.Create()`, открывает полный доступ по чтению и записи всем пользователям.

Также обратите внимание, что после окончания работы с текущим объектом `FileStream` необходимо обеспечить закрытие его дескриптора для освобождения внутренних неуправляемых ресурсов потока. Учитывая, что `FileStream` реализует интерфейс `IDisposable`, можно использовать блок `using` и позволить компилятору генерировать логику завершения (подробности ищите в главе 8):

```
var fileName =
${"C${Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat"};
...

// Поместить файловый поток внутрь оператора using.
FileInfo f1 = new FileInfo(fileName);
using (FileStream fs1 = f1.Create())
{
    // Использовать объект FileStream...
}
f1.Delete();
```

На заметку! Почти все примеры в этой главе содержат операторы `using`. Можно также использовать новый синтаксис объявлений `using`, но было решено придерживаться операторов `using`, чтобы сосредоточить примеры на исследуемых компонентах `System.IO`.

Метод FileInfo.Open()

С помощью метода FileInfo.Open() можно открывать существующие файлы, а также создавать новые файлы с более высокой точностью представления, чем обеспечивает метод FileInfo.Create(), поскольку Open() обычно принимает несколько параметров для описания общей структуры файла, с которым будет производиться работа. В результате вызова Open() возвращается объект FileStream. Взгляните на следующий код:

```
var fileName =
    $"{Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat";
...
// Создать новый файл посредством FileInfo.Open().
FileInfo f2 = new FileInfo(fileName);
using(FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
    FileAccess.ReadWrite, FileShare.None))
{
    // Использовать объект FileStream...
}
f2.Delete();
```

Эта версия перегруженного метода Open() требует передачи трех параметров. Первый параметр указывает общий тип запроса ввода-вывода (например, создать новый файл, открыть существующий файл или дописать в файл), который представлен в виде перечисления FileMode (описание его членов приведено в табл. 20.5):

```
public enum FileMode
{
    CreateNew,
    Create,
    Open,
    OpenOrCreate,
    Truncate,
    Append
}
```

Таблица 20.5. Члены перечисления FileMode

Член	Описание
CreateNew	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, то генерируется исключение IOException
Create	Информирует операционную систему о необходимости создания нового файла. Если файл уже существует, тогда он будет перезаписан
Open	Открывает существующий файл. Если файл не существует, то генерируется исключение FileNotFoundException
OpenOrCreate	Открывает файл, если он существует; в противном случае создает новый
Truncate	Открывает файл и усекает его до нулевой длины
Append	Открывает файл, переходит в его конец и начинает операции записи (этот флаг может применяться лишь с потоками только для записи). Если файл не существует, тогда создается новый файл

Второй параметр метода Open() — значение перечисления FileAccess — служит для определения поведения чтения/записи лежащего в основе потока:

```
public enum FileAccess
{
    Read,
    Write,
    ReadWrite
}
```

Наконец, третий параметр метода Open() — значение перечисления FileShare — указывает, каким образом файл может совместно использоваться другими файловыми дескрипторами:

```
public enum FileShare
{
    None,
    Read,
    Write,
    ReadWrite,
    Delete,
    Inheritable
}
```

Методы FileInfo.OpenRead() и FileInfo.OpenWrite()

Метод FileInfo.Open() позволяет получить дескриптор файла в гибкой манере, но класс FileInfo также предлагает методы OpenRead() и OpenWrite(). Как и можно было ожидать, указанные методы возвращают подходящим образом сконфигурированный только для чтения или только для записи объект FileStream без необходимости в предоставлении значений разных перечислений. Подобно FileInfo.Create() и FileInfo.Open() методы OpenRead() и OpenWrite() возвращают объект FileStream.

Обратите внимание, что метод OpenRead() требует, чтобы файл существовал. Следующий код создает файл и затем закрывает объект FileStream, так что он может использоваться методом OpenRead():

```
f3.Create().Close();
```

Вот полный пример:

```
var fileName =
    $"{Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat";
...
// Получить объект FileStream с правами только для чтения.
FileInfo f3 = new FileInfo(fileName);
// Перед использованием OpenRead() файл должен существовать.
f3.Create().Close();
using(FileStream readOnlyStream = f3.OpenRead())
{
    // Использовать объект FileStream...
}
f3.Delete();

// Теперь получить объект FileStream с правами только для записи.
FileInfo f4 = new FileInfo(fileName);
```

```
using(FileStream writeOnlyStream = f4.OpenWrite())
{
    // Использовать объект FileStream...
}
f4.Delete();
```

Метод FileInfo.OpenText()

Еще одним членом типа FileInfo, связанным с открытием файлов, является OpenText(). В отличие от Create(), Open(), OpenRead() и OpenWrite() метод OpenText() возвращает экземпляр типа StreamReader, а не FileStream. Исходя из того, что на диске C: имеется файл по имени boot.ini, вот как получить доступ к его содержимому:

```
var fileName =
    $"{Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat";
...
// Получить объект StreamReader.
// Если вы работаете не на машине с Windows,
// тогда измените имя файла надлежащим образом.
FileInfo f5 = new FileInfo(fileName);
// Перед использованием OpenText() файл должен существовать.
f5.Create().Close();
using(StreamReader sreader = f5.OpenText())
{
    // Использовать объект StreamReader...
}
f5.Delete();
```

Вскоре вы увидите, что тип StreamReader предоставляет способ чтения символьных данных из лежащего в основе файла.

Методы FileInfo.CreateText() и FileInfo.AppendText()

Последними двумя методами, представляющими интерес в данный момент, являются CreateText() и AppendText(). Оба они возвращают объект StreamWriter:

```
var fileName =
    $"{Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat";
...
FileInfo f6 = new FileInfo(fileName);
using(StreamWriter swriter = f6.CreateText())
{
    // Использовать объект StreamWriter...
}
f6.Delete();

FileInfo f7 = new FileInfo(fileName);
using(StreamWriter swriterAppend = f7.AppendText())
{
    // Использовать объект StreamWriter...
}
f7.Delete();
```

Как и можно было ожидать, тип `StreamWriter` предлагает способ записи данных в связанный с ним файл.

Работа с типом `File`

В типе `File` определено несколько статических методов для предоставления функциональности, почти идентичной той, которая доступна в типе `FileInfo`. Подобно `FileInfo` тип `File` поддерживает методы `AppendText()`, `Create()`, `CreateText()`, `Open()`, `OpenRead()`, `OpenWrite()` и `OpenText()`. Во многих случаях типы `File` и `FileInfo` могут применяться взаимозаменяющими. Обратите внимание, что методы `OpenText()` и `OpenRead()` требуют существования файла. Чтобы взглянуть на тип `File` в действии, упростите приведенные ранее примеры использования типа `FileStream`, применив в каждом из них тип `File`:

```
var fileName =
    $"{C\Path.VolumeSeparatorChar}{Path.DirectorySeparatorChar}Test.dat";
...
// Использование File вместо FileInfo.
using (FileStream fs8 = File.Create(fileName))
{
    // Использовать объект FileStream...
}
File.Delete(fileName);

// Создать новый файл через File.Open().
using(FileStream fs9 = File.Open(fileName,
    FileMode.OpenOrCreate, FileAccess.ReadWrite,
    FileShare.None))
{
    // Использовать объект FileStream...
}

// Получить объект FileStream с правами только для чтения.
using(FileStream readOnlyStream = File.OpenRead(fileName))
{}
File.Delete(fileName);

// Получить объект FileStream с правами только для записи.
using(FileStream writeOnlyStream = File.OpenWrite(fileName))
{}

// Получить объект StreamReader.
using(StreamReader sreader = File.OpenText(fileName))
{}
File.Delete(fileName);

// Получить несколько объектов StreamWriter.
using(StreamWriter swriter = File.CreateText(fileName))
{}
File.Delete(fileName);

using(StreamWriter swriterAppend =
    File.AppendText(fileName))
{}
File.Delete(fileName);
```

Дополнительные члены типа `File`

Тип `File` также поддерживает несколько членов, описанных в табл. 20.6, которые могут значительно упростить процессы чтения и записи текстовых данных.

Таблица 20.6. Методы типа `File`

Метод	Описание
<code>ReadAllBytes()</code>	Открывает указанный файл, возвращает двоичные данные в виде массива байтов и закрывает файл
<code>ReadAllLines()</code>	Открывает указанный файл, возвращает символьные данные в виде массива строк и закрывает файл
<code>ReadAllText()</code>	Открывает указанный файл, возвращает символьные данные в виде объекта <code>System.String</code> и закрывает файл
<code>WriteAllBytes()</code>	Открывает указанный файл, записывает в него массив байтов и закрывает файл
<code>WriteAllLines()</code>	Открывает указанный файл, записывает в него массив строк и закрывает файл
<code>WriteAllText()</code>	Открывает указанный файл, записывает в него символьные данные из заданной строки и закрывает файл

Приведенные в табл. 20.6 методы типа `File` можно использовать для реализации чтения и записи пакетов данных посредством всего нескольких строк кода. Еще лучше то, что эти методы автоматически закрывают лежащий в основе файловый дескриптор. Например, следующий проект консольного приложения (по имени `SimpleFileIO`) сохраняет строковые данные в новом файле на диске C: (и читает их в память) с минимальными усилиями (здесь предполагается, что было импортировано пространство имен `System.IO`):

```
Console.WriteLine("***** Simple I/O with the File Type *****\n");
string[] myTasks = {
    "Fix bathroom sink", "Call Dave",
    "Call Mom and Dad", "Play Xbox One"};

// Записать все данные в файл на диске C:.
File.WriteAllLines(@"tasks.txt", myTasks);

// Прочитать все данные и вывести на консоль.
foreach (string task in File.ReadAllLines(@"tasks.txt"))
{
    Console.WriteLine("TODO: {0}", task);
}
Console.ReadLine();
File.Delete("tasks.txt");
```

Из продемонстрированного примера можно сделать вывод: когда необходимо быстро получить файловый дескриптор, тип `File` позволит сэкономить на объеме кодирования. Тем не менее, преимущество предварительного создания объекта `FileInfo` заключается в возможности сбора сведений о файле с применением членов абстрактного базового класса `FileSystemInfo`.

Абстрактный класс Stream

Вы уже видели много способов получения объектов `FileStream`, `StreamReader` и `StreamWriter`, но с использованием упомянутых типов нужно еще читать данные или записывать их в файл. Чтобы понять, как это делается, необходимо освоить концепцию потока. В мире манипуляций вводом-выводом *поток* (*stream*) представляет порцию данных, протекающую между источником и приемником. Потоки предоставляют общий способ взаимодействия с последовательностью байтов независимо от того, устройство какого рода (файл, сетевое подключение либо принтер) хранит или отображает байты.

Абстрактный класс `System.IO.Stream` определяет набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

На заметку! Концепция потока не ограничена файловым вводом-выводом. Естественно, библиотеки .NET Core предлагают потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Потомки класса `Stream` представляют данные в виде низкоуровневых потоков байтов; следовательно, работа непосредственно с низкоуровневыми потоками может оказаться не особенно понятной. Некоторые типы, производные от `Stream`, поддерживают позиционирование, которое означает процесс получения и корректировки текущей позиции в потоке. В табл. 20.7 приведено описание основных членов класса `Stream`, что помогает понять его функциональность.

Таблица 20.7. Члены абстрактного класса Stream

Член	Описание
<code>CanRead</code>	Определяют, поддерживает ли текущий поток чтение, поиск и/или запись
<code>CanWrite</code>	
<code>CanSeek</code>	
<code>Close()</code>	Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом <code>Dispose()</code> , поэтому закрытие потока функционально эквивалентно освобождению потока
<code>Flush()</code>	Обновляет лежащий в основе источник данных или хранилище текущим состоянием буфера и затем очищает буфер. Если поток не реализует буфер, то метод ничего не делает
<code>Length</code>	Возвращает длину потока в байтах
<code>Position</code>	Определяет текущую позицию в потоке
<code>Read()</code>	Читают последовательность байтов (или одиночный байт) из текущего потока и перемещают текущую позицию потока вперед на количество прочитанных байтов
<code>ReadByte()</code>	
<code>ReadAsync()</code>	
<code>Seek()</code>	Устанавливает позицию в текущем потоке
<code>SetLength()</code>	Устанавливает длину текущего потока
<code>Write()</code>	Записывают последовательность байтов (или одиночный байт) в текущий поток и перемещают текущую позицию вперед на количество записанных байтов
<code>WriteByte()</code>	
<code>WriteAsync()</code>	

Работа с типом FileStream

Класс `FileStream` предоставляет реализацию абстрактных членов `Stream` в менюре, подходящей для потоковой работы с файлами. Это элементарный поток; он может записывать или читать только одиночный байт или массив байтов. Однако напрямую взаимодействовать с членами типа `FileStream` вам придется нечасто. Взамен вы, скорее всего, будете применять разнообразные оболочки потоков, которые облегчают работу с текстовыми данными или типами .NET Core. Тем не менее, полезно поэкспериментировать с возможностями синхронного чтения/записи типа `FileStream`.

Пусть имеется новый проект консольного приложения под названием `FileStreamApp` (и в файле кода C# импортировано пространство имен `System.IO` и `System.Text`). Целью будет запись простого текстового сообщения в новый файл по имени `myMessage.dat`. Однако с учетом того, что `FileStream` может оперировать только с низкоуровневыми байтами, объект типа `System.String` придется закодировать в соответствующий байтовый массив. К счастью, в пространстве имен `System.Text` определен тип `Encoding`, предоставляющий члены, которые кодируют и декодируют строки в массивы байтов.

После кодирования байтовый массив сохраняется в файле с помощью метода `FileStream.Write()`. Чтобы прочитать байты обратно в память, понадобится сбросить внутреннюю позицию потока (посредством свойства `Position`) и вызвать метод `ReadByte()`. Наконец, на консоль выводится содержимое низкоуровневого байтового массива и декодированная строка. Ниже приведен полный код:

```
using System;
using System.IO;
using System.Text;
// Не забудьте импортировать пространства имен System.Text и System.IO.
Console.WriteLine("***** Fun with FileStreams *****\n");
// Получить объект FileStream.
using(FileStream fStream = File.Open("myMessage.dat", FileMode.Create))
{
    // Закодировать строку в виде массива байтов.
    string msg = "Hello!";
    byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);
    // Записать byte[] в файл.
    fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);
    // Сбросить внутреннюю позицию потока.
    fStream.Position = 0;
    // Прочитать byte[] из файла и вывести на консоль.
    Console.Write("Your message as an array of bytes: ");
    byte[] bytesFromFile = new byte[msgAsByteArray.Length];
    for (int i = 0; i < msgAsByteArray.Length; i++)
    {
        bytesFromFile[i] = (byte)fStream.ReadByte();
        Console.Write(bytesFromFile[i]);
    }
    // Вывести декодированное сообщение.
    Console.Write("\nDecoded Message: ");
    Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
    Console.ReadLine();
}
File.Delete("myMessage.dat");
```

В приведенном примере не только производится наполнение файла данными, но также демонстрируется основной недостаток прямой работы с типом `FileStream`: необходимость оперирования низкоуровневыми байтами. Другие производные от `Stream` типы работают в похожей манере. Например, чтобы записать последовательность байтов в область памяти, понадобится создать объект `MemoryStream`.

Как упоминалось ранее, в пространстве имен `System.IO` доступно несколько типов для средств чтения и записи, которые инкапсулируют детали работы с типами, производными от `Stream`.

Работа с типами `StreamWriter` и `StreamReader`

Классы `StreamWriter` и `StreamReader` удобны всякий раз, когда нужно читать или записывать символьные данные (например, строки). Оба типа по умолчанию работают с символами Unicode; тем не менее, это можно изменить за счет представления должным образом сконфигурированной ссылки на объект `System.Text.Encoding`. Чтобы не усложнять пример, предположим, что стандартная кодировка Unicode вполне устраивает.

Класс `StreamReader` является производным от абстрактного класса по имени `TextReader`, как и связанный с ним тип `StringReader` (обсуждается далее в главе). Базовый класс `TextReader` предоставляет каждому из своих наследников ограниченный набор функциональных средств, в частности возможность читать и "заглядывать" в символьный поток.

Класс `StreamWriter` (а также `StringWriter`, который будет рассматриваться позже) порожден от абстрактного базового класса по имени `TextWriter`, в котором определены члены, позволяющие производным типам записывать текстовые данные в текущий символьный поток.

Чтобы содействовать пониманию основных возможностей записи в классах `StreamWriter` и `StringWriter`, в табл. 20.8 перечислены основные члены абстрактного базового класса `TextWriter`.

Таблица 20.8. Основные члены `TextWriter`

Член	Описание
<code>Close()</code>	Этот метод закрывает средство записи и освобождает все связанные с ним ресурсы. В процессе буфер автоматически сбрасывается (и снова этот член функционально эквивалентен методу <code>Dispose()</code>)
<code>Flush()</code>	Этот метод очищает все буфера текущего средства записи и записывает все буферизованные данные на лежащее в основе устройство; однако он не закрывает средство записи
<code>NewLine</code>	Это свойство задает константу новой строки для унаследованного класса средства записи. По умолчанию ограничителем строки в Windows является возврат каретки, за которым следует перевод строки (<code>\r\n</code>)
<code>Write()</code>	Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки
<code>WriteLine()</code>	Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

На заметку! Вероятно, последние два члена класса `TextWriter` покажутся знакомыми.

Вспомните, что тип `System.Console` имеет члены `Write()` и `WriteLine()`, которые выталкивают текстовые данные на стандартное устройство вывода. В действительности свойство `Console.In` является оболочкой для объекта `TextWriter`, а `Console.Out` — для `TextWriter`.

Производный класс `StreamWriter` предоставляет подходящую реализацию методов `Write()`, `Close()` и `Flush()`, а также определяет дополнительное свойство `AutoFlush`. Установка этого свойства в `true` заставляет `StreamWriter` выталкивать данные при каждой операции записи. Имейте в виду, что за счет установки `AutoFlush` в `false` можно достичь более высокой производительности, но по завершении работы с объектом `StreamWriter` должен быть вызван метод `Close()`.

Запись в текстовый файл

Чтобы увидеть класс `StreamWriter` в действии, создайте новый проект консольного приложения по имени `StreamWriterReaderApp` и импортируйте пространства имен `System.IO` и `System.Text`. В показанном ниже коде с помощью метода `File.CreateText()` создается новый файл `reminders.txt` внутри текущего каталога выполнения. С применением полученного объекта `StreamWriter` в новый файл будут добавляться текстовые данные.

```
using System;
using System.IO;
using System.Text;

Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
// Получить объект StreamWriter и записать строковые данные.
using(StreamWriter writer = File.CreateText("reminders.txt"))
{
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
    for(int i = 0; i < 10; i++)
    {
        writer.Write(i + " ");
    }
    // Вставить новую строку.
    writer.Write(writer.NewLine);
}

Console.WriteLine("Created file and wrote some thoughts...");
Console.ReadLine();
// File.Delete("reminders.txt");
```

После выполнения программы можете просмотреть содержимое созданного файла, который будет находиться в корневом каталоге проекта (Visual Studio Code) или в подкаталоге `bin\Debug\net5.0` (Visual Studio). Причина в том, что при вызове `CreateText()` вы не указали абсолютный путь, а стандартным местоположением является текущий каталог выполнения сборки.

Чтение из текстового файла

Далее вы научитесь программно читать данные из файла, используя соответствующий тип `StreamReader`. Вспомните, что `StreamReader` является производным от абстрактного класса `TextReader`, который предлагает функциональность, описанную в табл. 20.9.

Таблица 20.9. Основные члены `TextReader`

Член	Описание
<code>Peek()</code>	Возвращает следующий доступный символ (выраженный в виде целого числа), не изменяя текущей позиции средства чтения. Значение <code>-1</code> указывает на достижение конца потока
<code>Read()</code>	Читает данные из входного потока
<code>ReadBlock()</code>	Читает указанное максимальное количество символов из текущего потока и записывает данные в буфер, начиная с заданного индекса
<code>ReadLine()</code>	Читает строку символов из текущего потока и возвращает данные в виде строки (строка <code>null</code> указывает на признак конца файла)
<code>ReadToEnd()</code>	Читает все символы от текущей позиции до конца потока и возвращает их в виде единственной строки

Расширьте текущий пример приложения с целью применения класса `StreamReader`, чтобы в нем можно было читать текстовые данные из файла `reminders.txt`:

```
Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
...
// Прочитать данные из файла.
Console.WriteLine("Here are your thoughts:\n");
using(StreamReader sr = File.OpenText("reminders.txt"))
{
    string input = null;
    while ((input = sr.ReadLine()) != null)
    {
        Console.WriteLine (input);
    }
}
Console.ReadLine();
```

После запуска программы в окне консоли отобразятся символьные данные из файла `reminders.txt`.

Прямое создание объектов типа `StreamWriter/StreamReader`

Один из запутывающих аспектов работы с типами пространства имен `System.IO` связан с тем, что идентичных результатов часто можно добиться с использованием разных подходов. Например, ранее вы уже видели, что метод `CreateText()` позволяет получить объект `StreamWriter` с типом `File` или `FileInfo`. Вообще говоря, есть еще один способ работы с объектами `StreamWriter` и `StreamReader`: создание их напрямую. Скажем, текущее приложение можно было бы переделать следующим образом:

```
Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
// Получить объект StreamWriter и записать строковые данные.
using(StreamWriter writer = new StreamWriter("reminders.txt"))
{
}
...
// Прочитать данные из файла.
using(StreamReader sr = new StreamReader("reminders.txt"))
{
}
...
```

Несмотря на то что существование такого количества на первый взгляд одинаковых подходов к файловому вводу-выводу может сбивать с толку, имейте в виду, что конечным результатом является высокая гибкость. Теперь, когда вам известно, как перемещать символьные данные в файл и из файла с применением классов StreamWriter и StreamReader, давайте займемся исследованием роли классов StringWriter и StreamReader.

Работа с типами **StringWriter** и **StringReader**

Классы StringWriter и StreamReader можно использовать для трактовки текстовой информации как потока символов в памяти. Это определенно может быть полезно, когда нужно добавить символьную информацию к лежащему в основе буферу. Для иллюстрации в следующем проекте консольного приложения (StringReaderWriterApp) блок строковых данных записывается в объект StringWriter вместо файла на локальном жестком диске (не забудьте импортировать пространства имен System.IO и System.Text):

```
using System;
using System.IO;
using System.Text;

Console.WriteLine("***** Fun with StringWriter / StreamReader *****\n");
// Создать объект StringWriter и записать символьные данные в память.
using(StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    // Получить копию содержимого (хранящегося в строке) и вывести на консоль
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
}
Console.ReadLine();
```

Классы StringWriter и StreamReader порождены от одного и того же базового класса (TextWriter), поэтому логика записи похожа. Тем не менее, с учетом природы StringWriter вы должны также знать, что данный класс позволяет применять метод GetStringBuilder() для извлечения объекта System.Text.StringBuilder:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
```

```
// Получить внутренний объект StringBuilder.
StringBuilder sb = strWriter.GetStringBuilder();
sb.Insert(0, "Hey!! ");
Console.WriteLine("-> {0}", sb.ToString());
sb.Remove(0, "Hey!! ".Length);
Console.WriteLine("-> {0}", sb.ToString());
}
```

Когда необходимо прочитать из потока строковые данные, можно использовать соответствующий тип `StringReader`, который (вполне ожидаемо) функционирует идентично `StreamReader`. Фактически класс `StringReader` лишь переопределяет унаследованные члены, чтобы выполнять чтение из блока символьных данных, а не из файла:

```
using (StringWriter strWriter = new StringWriter())
{
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

    // Читать данные из объекта StringWriter.
    using (StringReader strReader = new StringReader(strWriter.ToString()))
    {
        string input = null;
        while ((input = strReader.ReadLine()) != null)
        {
            Console.WriteLine(input);
        }
    }
}
```

Работа с типами `BinaryWriter` и `BinaryReader`

Последним набором классов средств чтения и записи, которые рассматриваются в настоящем разделе, являются `BinaryWriter` и `BinaryReader`; они оба унаследованы прямо от `System.Object`. Типы `BinaryWriter` и `BinaryReader` позволяют читать и записывать в поток дискретные типы данных в компактном двоичном формате. В классе `BinaryWriter` определен многократно перегруженный метод `Write()`, предназначенный для помещения некоторого типа данных в поток. Помимо `Write()` класс `BinaryWriter` предоставляет дополнительные члены, которые позволяют получать или устанавливать объекты производных от `Stream` типов; кроме того, класс `BinaryWriter` также предлагает поддержку произвольного доступа к данным (табл. 20.10).

Таблица 20.10. Основные члены `BinaryWriter`

Член	Описание
<code>BaseStream</code>	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом <code>BinaryWriter</code>
<code>Close()</code>	Этот метод закрывает двоичный поток
<code>Flush()</code>	Этот метод выталкивает буфер двоичного потока
<code>Seek()</code>	Этот метод устанавливает позицию в текущем потоке
<code>Write()</code>	Этот метод записывает значение в текущий поток

Класс `BinaryReader` дополняет функциональность класса `BinaryWriter` членами, описанными в табл. 20.11.

Таблица 20.11. Основные члены `BinaryReader`

Член	Описание
<code>BaseStream</code>	Это свойство только для чтения обеспечивает доступ к лежащему в основе потоку, используемому с объектом <code>BinaryReader</code>
<code>Close()</code>	Этот метод закрывает двоичный поток
<code>PeekChar()</code>	Этот метод возвращает следующий доступный символ без перемещения текущей позиции потока
<code>Read()</code>	Этот метод читает заданный набор байтов или символов и сохраняет их во входном массиве
<code>ReadXXXX()</code>	В классе <code>BinaryReader</code> определены многочисленные методы чтения, которые извлекают из потока объекты различных типов (<code>ReadBoolean()</code> , <code>ReadByte()</code> , <code>ReadInt32()</code> и т.д.)

В показанном далее примере (проект консольного приложения по имени `BinaryWriterReader` с оператором `using` для `System.IO`) в файл `*.dat` записываются данные нескольких типов:

```
using System;
using System.IO;

Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");
// Открыть средство двоичной записи в файл.
FileInfo f = new FileInfo("BinFile.dat");
using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
{
    // Вывести на консоль тип BaseStream
    // (System.IO.FileStream в этом случае).
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);

    // Создать некоторые данные для сохранения в файле.
    double aDouble = 1234.67;
    int anInt = 34567;
    string aString = "A, B, C";

    // Записать данные.
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
    Console.WriteLine("Done!");
    Console.ReadLine();
}
```

Обратите внимание, что объект `FileStream`, возвращенный методом `FileInfo.OpenWrite()`, передается конструктору типа `BinaryWriter`. Применение такого приема облегчает организацию потока по уровням перед записью данных. Конструктор класса `BinaryWriter` принимает любой тип, производный от `Stream` (например, `FileStream`, `MemoryStream` или `BufferedStream`). Таким образом, запись двоичных данных в память сводится просто к использованию допустимого объекта `MemoryStream`.

Для чтения данных из файла BinFile.dat в классе BinaryReader предлагается несколько способов. Ниже для извлечения каждой порции данных из файлового потока вызываются разнообразные члены, связанные с чтением:

```
...
FileInfo f = new FileInfo("BinFile.dat");
...
// Читать двоичные данные из потока.
using(BinaryReader br = new BinaryReader(f.OpenRead()))
{
    Console.WriteLine(br.ReadDouble());
    Console.WriteLine(br.ReadInt32());
    Console.WriteLine(br.ReadString());
}
Console.ReadLine();
```

Программное слежение за файлами

Теперь, когда вы знаете, как применять различные средства чтения и записи, давайте займемся исследованием роли класса FileSystemWatcher, который полезен, когда требуется программно отслеживать состояние файлов в системе. В частности, с помощью FileSystemWatcher можно организовать мониторинг файлов на предмет любых действий, указываемых значениями перечисления System.IO.NotifyFilters:

```
public enum NotifyFilters
{
    Attributes, CreationTime,
   DirectoryName, FileName,
    LastAccess, LastWrite,
    Security, Size
}
```

Чтобы начать работу с типом FileSystemWatcher, в свойстве Path понадобится указать имя (и местоположение) каталога, содержащего файлы, которые нужно отслеживать, а в свойстве Filter — расширения отслеживаемых файлов.

В настоящий момент можно выбрать обработку событий Changed, Created и Deleted, которые функционируют в сочетании с делегатом FileSystemEventHandler. Этот делегат может вызывать любой метод, соответствующий следующей сигнатуре:

```
// Делегат FileSystemEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

Событие Renamed может быть также обработано с использованием делегата RenamedEventHandler, который позволяет вызывать методы с такой сигнатурой:

```
// Делегат RenamedEventHandler должен указывать
// на методы, соответствующие следующей сигнатуре.
void MyRenamedHandler(object source, RenamedEventArgs e)
```

В то время как для обработки каждого события можно применять традиционный синтаксис делегатов/событий, вы определенно будете использовать синтаксис лямбда-выражений.

44 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

Давайте взглянем на процесс слежения за файлом. Показанный ниже проект консольного приложения (MyDirectoryWatcher с оператором using для System.IO) наблюдает за файлами *.txt в каталоге bin\debug\net5.0 и выводит на консоль сообщения, когда происходит их создание, удаление, модификация и переименование:

```
using System;
using System.IO;

Console.WriteLine("***** The Amazing File Watcher App *****\n");

// Установить путь к каталогу, за которым нужно наблюдать.
FileSystemWatcher watcher = new FileSystemWatcher();
try
{
    watcher.Path = @".";
}
catch (ArgumentException ex)
{
    Console.WriteLine(ex.Message);
    return;
}

// Указать цели наблюдения.
watcher.NotifyFilter = NotifyFilters.LastAccess
| NotifyFilters.LastWrite
| NotifyFilters.FileName
| NotifyFilters.DirectoryName;

// Следить только за текстовыми файлами.
watcher.Filter = "*.txt";

// Добавить обработчики событий.
// Указать, что будет происходить при изменении,
// создании или удалении файла.
watcher.Changed += (s, e) =>
    Console.WriteLine($"File: {e.FullPath} {e.ChangeType}!");
watcher.Created += (s, e) =>
    Console.WriteLine($"File: {e.FullPath} {e.ChangeType}!");
watcher.Deleted += (s, e) =>
    Console.WriteLine($"File: {e.FullPath} {e.ChangeType}!");

// Указать, что будет происходить при переименовании файла.
watcher.Renamed += (s, e) =>
    Console.WriteLine($"File: {e.OldFullPath} renamed to {e.FullPath}");

// Начать наблюдение за каталогом.
watcher.EnableRaisingEvents = true;

// Ожидать от пользователя команды завершения программы.
Console.WriteLine(@"Press 'q' to quit app.");
                // Нажмите q, чтобы завершить приложение.

// Сгенерировать несколько событий.
using (var sw = File.CreateText("Test.txt"))
{
    sw.Write("This is some text");
}
File.Move("Test.txt", "Test2.txt");
File.Delete("Test2.txt");
while(Console.Read() != 'q');
```

При запуске данной программы последние ее строки будут создавать, изменять, переименовывать и затем удалять текстовый файл, попутно генерируя события. Кроме того, вы можете перейти в каталог bin\debug\net5.0 и поработать с файлами (имеющими расширение *.txt), что приведет к инициированию дополнительных событий.

```
***** The Amazing File Watcher App *****
Press 'q' to quit app.
File: .\Test.txt Created!
File: .\Test.txt Changed!
File: .\Test.txt renamed to .\Test2.txt
File: .\Test2.txt Deleted!
```

На этом знакомство с фундаментальными операциями ввода-вывода, предлагаемыми платформой .NET Core, завершено. Вы наверняка будете применять все продемонстрированные приемы во многих приложениях. Вдобавок вы обнаружите, что службы сериализации объектов способны значительно упростить задачу сохранения больших объемов данных.

Понятие сериализации объектов

Термин *сериализация* описывает процесс сохранения (и возможно передачи) состояния объекта в потоке (например, файловом потоке или потоке в памяти). Сохраненная последовательность данных содержит всю информацию, необходимую для воссоздания (или *десериализации*) открытого состояния объекта с целью последующего использования. Применение такой технологии делает тривиальным сохранение крупных объемов данных (в разнообразных форматах). Во многих случаях сохранение данных приложения с использованием служб сериализации дает в результате меньше кода, чем с применением средств чтения/записи из пространства имен System.IO.

Например, пусть требуется создать настольное приложение с графическим пользовательским интерфейсом, которое должно предоставлять конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.д.). Для этого можно определить класс по имени UserPrefs и инкапсулировать в нем около двадцати полей данных. В случае использования типа System.IO.BinaryWriter пришлось бы *вручную* сохранять каждое поле объекта UserPrefs. Подобным же образом при загрузке данных из файла обратно в память понадобилось бы применять класс System.IO.BinaryReader и снова *вручную* читать каждое значение, чтобы повторно сконфигурировать новый объект UserPrefs.

Все это выполнимо, но вы можете сэкономить значительное время за счет использования сериализации XML (eXtensible Markup Language — расширяемый язык разметки) или JSON (JavaScript Object Notation — запись объектов JavaScript). Каждый из указанных форматов состоит из пар “имя-значение”, позволяя представлять открытое состояние объекта в одиночном блоке текста, который можно потреблять между платформами и языками программирования. В итоге полное открытое состояние объекта может быть сохранено с помощью лишь нескольких строк кода.

На заметку! Применение типа BinaryFormatter (<https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-5.0>), который рассматривался в предшествующих изданиях книги, сопряжено с высоким риском в плане безопасности, так что от него следует немедленно отказаться. Более защищенные альтернативы предусматривают использование классов BinaryReader/BinaryWriter для XML/JSON.

Сериализация объектов .NET Core упрощает сохранение объектов, но ее внутренний процесс довольно сложен. Например, когда объект сохраняется в потоке, все ассоциированные с ним открытые данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализируются. Следовательно, при попытке сериализации производного класса в игру вступают также все открытые данные по цепочке наследования. Вы увидите, что для представления множества взаимосвязанных объектов используется граф объектов.

Наконец, имейте в виду, что граф объектов может быть сохранен в любом типе, производном от `System.IO.Stream`. Важно лишь то, чтобы последовательность данных корректно представляла состояние объектов внутри графа.

Роль графов объектов

Как упоминалось ранее, среда CLR будет учитывать все связанные объекты, чтобы обеспечить корректное сохранение данных, когда объект сериализуется. Такой набор связанных объектов называется *графом объектов*. Графы объектов предоставляют простой способ документирования взаимосвязи между множеством элементов. Следует отметить, что графы объектов не обозначают отношения “является” и “имеет” объектно-ориентированного программирования. Взамен стрелки в графе объектов можно трактовать как “требует” или “зависит от”.

Каждый объект в графе получает уникальное числовое значение. Важно помнить, что числа, присвоенные объектам в графе, являются произвольными и не имеют никакого смысла для внешнего мира. После того как всем объектам назначены числовые значения, граф объектов может записывать набор зависимостей для каждого объекта.

В качестве примера предположим, что создано множество классов, которые моделируют автомобили. Существует базовый класс по имени `Car`, который “имеет” класс `Radio`. Другой класс по имени `JamesBondCar` расширяет базовый тип `Car`.

На рис. 20.1 показан возможный график объектов, моделирующий такие отношения.

При чтении графов объектов для описания соединяющих стрелок можно использовать выражение “зависит от” или “ссылается на”. Таким образом, на рис. 20.1 видно, что класс `Car` ссылается на класс `Radio` (учитывая отношение “имеет”), `JamesBondCar` ссылается на `Car` (из-за отношения “является”), а также на `Radio` (поскольку наследует эту защищенную переменную-член).

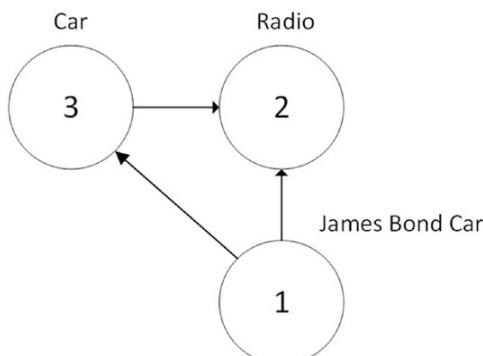


Рис. 20.1. Простой график объектов

Разумеется, исполняющая среда не рисует картинки в памяти для представления графа связанных объектов. Взамен отношение, показанное на рис. 20.1, представляется математической формулой, которая выглядит следующим образом:

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

Проанализировав формулу, вы заметите, что объект 3 (Car) имеет зависимость от объекта 2 (Radio). Объект 2 (Radio) — “одинокий волк”, которому никто не нужен. Наконец, объект 1 (JamesBondCar) имеет зависимость от объекта 3, а также от объекта 2. В любом случае при сериализации или десериализации экземпляра JamesBondCar граф объектов гарантирует, что типы Radio и Car тоже примут участие в процессе.

Привлекательность процесса сериализации заключается в том, что граф, представляющий отношения между объектами, устанавливается автоматически “за кулисами”. Как будет показано позже в главе, при желании в конструирование графа объектов можно вмешиваться, настраивая процесс сериализации с применением атрибутов и интерфейсов.

Создание примеров типов и написание операторов верхнего уровня

Создайте новый проект консольного приложения .NET 5 по имени SimpleSerialize. Добавьте в проект новый файл класса под названием Radio.cs со следующим кодом:

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text.Json.Serialization;
using System.Xml;
using System.Xml.Serialization;

namespace SimpleSerialize
{
    public class Radio
    {
        public bool HasTweeters;
        public bool HasSubWoofers;
        public List<double> StationPresets;
        public string RadioId = "XF-552RR6";
        public override string ToString()
        {
            var presets = string.Join(",",
                StationPresets.Select(i => i.ToString()).ToList());
            return $"HasTweeters:{HasTweeters}
                    HasSubWoofers:{HasSubWoofers} Station
                    Presets:{presets}";
        }
    }
}
```

Добавьте еще один файл класса по имени Car.cs и приведите его содержимое к такому виду:

```
using System;
using System.Text.Json.Serialization;
```

48 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
using System.Xml;
using System.Xml.Serialization;
namespace SimpleSerialize
{
    public class Car
    {
        public Radio TheRadio = new Radio();
        public bool IsHatchBack;
        public override string ToString()
            => $"IsHatchback:{IsHatchBack} Radio:{TheRadio.ToString()}";
    }
}
```

Затем добавьте очередной файл класса по имени JamesBondCar.cs и поместите в него следующий код:

```
using System;
using System.Text.Json.Serialization;
using System.Xml;
using System.Xml.Serialization;
namespace SimpleSerialize
{
    public class JamesBondCar : Car
    {
        public bool CanFly;
        public bool CanSubmerge;
        public override string ToString()
            => $"CanFly:{CanFly}, CanSubmerge:{CanSubmerge} {base.ToString()}";
    }
}
```

Ниже показан код финального файла класса Person.cs:

```
using System;
using System.Text.Json.Serialization;
using System.Xml;
using System.Xml.Serialization;
namespace SimpleSerialize
{
    public class Person
    {
        // Открытое поле.
        public bool IsAlive = true;
        // Закрытое поле.
        private int PersonAge = 21;
        // Открытое свойство/закрытые данные.
        private string _fName = string.Empty;
        public string FirstName
        {
            get { return _fName; }
            set { _fName = value; }
        }
        public override string ToString() =>
            $"IsAlive:{IsAlive} FirstName:{FirstName} Age:{PersonAge}";
    }
}
```

В заключение модифицируйте содержимое файла Program.cs, добавив следующий стартовый код:

```
using System.Collections.Generic;
using System.IO;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Xml;
using System.Xml.Serialization;
using SimpleSerialize;

Console.WriteLine("***** Fun with Object Serialization *****\n");
// Создать объект JamesBondCar и установить состояние.
JamesBondCar jbc = new()
{
    CanFly = true,
    CanSubmerge = false,
    TheRadio = new()
    {
        StationPresets = new() {89.3, 105.1, 97.1},
        HasTweeters = true
    }
};
Person p = new()
{
    FirstName = "James",
    IsAlive = true
};
```

Итак, все готово для того, чтобы приступить к исследованию сериализации XML и JSON.

Сериализация и десериализация с помощью XmlSerializer

Пространство имен `System.Xml` предоставляет класс `System.Xml.Serialization.XmlSerializer`. Этот форматер можно применять для сохранения открытого состояния заданного объекта в виде чистой XML-разметки. Важно отметить, что `XmlSerializer` требует объявления типа, который будет сериализоваться (или десериализоваться).

Управление генерацией данных XML

Если у вас есть опыт работы с технологиями XML, то вы знаете, что часто важно гарантировать соответствие данных внутри документа XML набору правил, которые устанавливают действительность данных. Понятие действительного документа XML не имеет никакого отношения к синтаксической правильности элементов XML (вроде того, что все открывающие элементы должны иметь закрывающие элементы). Действительные документы отвечают согласованным правилам форматирования (например, поле X должно быть выражено в виде атрибута, но не подэлемента), которые обычно задаются посредством схемы XML или файла определения типа документа (Document-Type Definition — DTD).

По умолчанию класс `XmlSerializer` сериализирует все открытые поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса `XmlSerializer`, необходимо декорировать

типы любым количеством дополнительных атрибутов .NET Core из пространства имен `System.Xml.Serialization`. В табл. 20.12 описаны некоторые (но не все) атрибуты .NET Core, влияющие на способ кодирования данных XML в потоке.

Таблица 20.12. Избранные атрибуты из пространства имен `System.Xml.Serialization`

Атрибут .NET Core	Описание
<code>[XmlAttribute]</code>	Этот атрибут .NET Core можно применять к полю или свойству для сообщения <code>XmlSerializer</code> о необходимости сериализовать данные как атрибут XML (а не как подэлемент)
<code>[XmlElement]</code>	Поле или свойство будет сериализовано как элемент XML, именованный по вашему выбору
<code>[XmlEnum]</code>	Этот атрибут предоставляет имя элемента члена перечисления
<code>[XmlRoot]</code>	Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и имя элемента)
<code>[XmlText]</code>	Свойство или поле будет сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)
<code>[XmlType]</code>	Этот атрибут предоставляет имя и пространство имен типа XML

Разумеется, для управления тем, как `XmlSerializer` генерирует результирующий XML-документ, можно использовать многие другие атрибуты .NET Core. Полные сведения ищите в описании пространства имен `System.Xml.Serialization` в документации по .NET Core.

На заметку! Класс `XmlSerializer` требует, чтобы все сериализуемые типы в графе объектов поддерживали стандартный конструктор (поэтому обязательно добавьте его обратно, если вы определяете специальные конструкторы).

Сериализация объектов с использованием `XmlSerializer`

Добавьте в свой файл `Program.cs` следующую локальную функцию:

```
static void SaveAsXmlFormat<T>(T objGraph, string fileName)
{
    // В конструкторе XmlSerializer должен быть объявлен тип.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(T));
    using (Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        xmlFormat.Serialize(fStream, objGraph);
    }
}
```

Добавьте к операторам верхнего уровня такой код:

```
SaveAsXmlFormat(jbc, "CarData.xml");
Console.WriteLine("=> Saved car in XML format!");

SaveAsXmlFormat(p, "PersonData.xml");
Console.WriteLine("=> Saved person in XML format!");
```

Заглянув внутрь сгенерированного файла `CarData.xml`, вы обнаружите в нем показанные ниже XML-данные:

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd=
    "http://www.w3.org/2001/XMLSchema" xmlns="http://www.MyCompany.com">
  <TheRadio>
    <HasTweeters>true</HasTweeters>
    <HasSubWoofers>false</HasSubWoofers>
    <StationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </StationPresets>
    <RadioId>XF-552RR6</RadioId>
  </TheRadio>
  <IsHatchBack>false</IsHatchBack>
  <CanFly>true</CanFly>
  <CanSubmerge>false</CanSubmerge>
</JamesBondCar>
```

Если вы хотите указать специальное пространство имен XML, которое уточняет `JamesBondCar` и кодирует значения `canFly` и `canSubmerge` в виде атрибутов XML, тогда модифицируйте определение класса `JamesBondCar` следующим образом:

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  [XmlAttribute]
  public bool CanFly;
  [XmlAttribute]
  public bool CanSubmerge;
  ...
}
```

Вот как будет выглядеть результирующий XML-документ (обратите внимание на открывающий элемент `<JamesBondCar>`):

```
<?xml version="1.0""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  CanFly="true" CanSubmerge="false" xmlns="http://www.MyCompany.com">
  ...
</JamesBondCar>
```

Иследуйте содержимое файла `PersonData.xml`:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <IsAlive>true</IsAlive>
  <FirstName>James</FirstName>
</Person>
```

Важно отметить, что свойство `PersonAge` не сериализуется в XML. Это подтверждает, что сериализация XML учитывает только открытые свойства и поля.

Сериализация коллекций объектов

Теперь, когда вы видели, каким образом сохранять одиничный объект в потоке, давайте посмотрим, как сохранить набор объектов. Создайте локальную функцию, которая инициализирует список объектов JamesBondCar и сериализует его в XML:

```
static void SaveListOfCarsAsXml()
{
    // Сохранить список List<T> объектов JamesBondCar.
    List<JamesBondCar> myCars = new()
    {
        new JamesBondCar{CanFly = true, CanSubmerge = true},
        new JamesBondCar{CanFly = true, CanSubmerge = false},
        new JamesBondCar{CanFly = false, CanSubmerge = true},
        new JamesBondCar{CanFly = false, CanSubmerge = false},
    };
    using (Stream fStream = new FileStream("CarCollection.xml",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
        xmlFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars!");
}
```

Наконец, добавьте следующую строку, чтобы задействовать новую функцию:

```
SaveListOfCarsAsXml();
```

Десериализация объектов и коллекций объектов

Десериализация XML буквально противоположна сериализации объектов (и коллекций объектов). Рассмотрим показанную далее локальную функцию для десериализации XML-разметки обратно в граф объектов. И снова обратите внимание, что тип, с которым нужно работать, должен быть передан конструктору XmlSerializer:

```
static T ReadAsXmlFormat<T>(string fileName)
{
    // Создать типизированный экземпляр класса XmlSerializer.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(T));
    using (Stream fStream = new FileStream(fileName, FileMode.Open))
    {
        T obj = default;
        obj = (T)xmlFormat.Deserialize(fStream);
        return obj;
    }
}
```

Добавьте к операторам верхнего уровня следующий код, чтобы восстановить XML-разметку обратно в объекты (или списки объектов):

```
JamesBondCar savedCar = ReadAsXmlFormat<JamesBondCar>("CarData.xml");
Console.WriteLine("Original Car: {0}", savedCar.ToString());
Console.WriteLine("Read Car: {0}", savedCar.ToString());

List<JamesBondCar> savedCars =
    ReadAsXmlFormat<List<JamesBondCar>>("CarCollection.xml");
```

Сериализация и десериализация с помощью `System.Text.Json`

В пространстве имен `System.Text.Json` имеется класс `System.Text.Json.JsonSerializer`, который вы можете использовать для сохранения открытого состояния заданного объекта как данных JSON.

Управление генерацией данных JSON

По умолчанию `JsonSerializer` сериализирует все открытые свойства в виде пар “имя-значение” в формате JSON, применяя такие же имена (и регистр символов), как у имен свойств объекта. Вы можете управлять многими аспектами процесса сериализации с помощью наиболее часто используемых атрибутов, перечисленных в табл. 20.13.

Таблица 20.13. Избранные атрибуты из пространства имен `System.Text.Json.Serialization`

Атрибут .NET Core	Описание
<code>[JsonIgnore]</code>	Свойство будет проигнорировано
<code>[JsonInclude]</code>	Член будет включен
<code>[JsonPropertyName]</code>	Задает имя свойства, которое будет применяться при сериализации/десериализации члена. Обычно используется для устранения проблем, связанных с регистром символов
<code>[JsonConstructor]</code>	Указывает конструктор, который должен применяться при десериализации данных JSON обратно в граф объектов

Сериализация объектов с использованием `JsonSerializer`

Класс `JsonSerializer` содержит статические методы `Serialize()`, применяемые для преобразования объектов .NET Core (включая графы объектов) в строковое представление открытых свойств. Данные представляются как пары “имя-значение” в формате JSON. Добавьте в файл `Program.cs` показанную ниже локальную функцию:

```
static void SaveAsJsonFormat<T>(T objGraph, string fileName)
{
    File.WriteAllText(fileName,
        System.Text.Json.JsonSerializer.Serialize(objGraph));
}
```

Добавьте к своим операторам верхнего уровня следующий код:

```
SaveAsJsonFormat(jbc, "CarData.json");
Console.WriteLine("> Saved car in JSON format!");

SaveAsJsonFormat(p, "PersonData.json");
Console.WriteLine("> Saved person in JSON format!');
```

Когда вы будете исследовать файлы JSON, вас может удивить тот факт, что файл `CarData.json` пуст (не считая пары фигурных скобок), а файл `PersonData.json` содержит только значение `Firstname`. Причина в том, что `JsonSerializer` по умолчанию записывает только открытые свойства, но не открытые поля. Проблема решается в следующем разделе.

Включение полей

Включить открытые поля в генерируемые данные JSON можно двумя способами. Первый способ предусматривает использование класса `JsonSerializerOptions` для сообщения `JsonSerializer` о необходимости включить все поля. Второй способ предполагает модификацию классов за счет добавления атрибута `[JsonInclude]` к каждому открытому полю, которое должно быть включено в вывод JSON. Обратите внимание, что при первом способе (применение `JsonSerializationOptions`) будут включаться *все* открытые поля в графе объектов. Чтобы исключить отдельные открытые поля с использованием такого приема, вам придется использовать для этих полей атрибут `JsonIgnore`.

Модифицируйте метод `SaveAsJsonFormat()`, как показано ниже:

```
static void SaveAsJsonFormat<T>(T objGraph, string fileName)
{
    var options = new JsonSerializerOptions
    {
        IncludeFields = true,
    };
    File.WriteAllText(fileName,
        System.Text.Json.JsonSerializer.Serialize(objGraph, options));
}
```

Вместо применения класса `JsonSerializerOptions` того же результата можно достичь, обновив все открытые поля в примерах классов следующим образом (имейте в виду, что вы можете оставить в классах атрибуты `Xml` и они не будут помехой `JsonSerializer`):

```
// Radio.cs
public class Radio
{
    [JsonInclude]
    public bool HasTweeters;
    [JsonInclude]
    public bool HasSubWoofers;
    [JsonInclude]
    public List<double> StationPresets;
    [JsonInclude]
    public string RadioId = "XF-552RR6";
    ...
}

// Car.cs
public class Car
{
    [JsonInclude]
    public Radio TheRadio = new Radio();
    [JsonInclude]
    public bool IsHatchBack;
    ...
}

// JamesBondCar.cs
public class JamesBondCar : Car
{
```

```
[XmlAttribute]
[JsonInclude]
public bool CanFly;
[XmlAttribute]
[JsonInclude]
public bool CanSubmerge;
...
}

// Person.cs
public class Person
{
    // Открытое поле.
    [JsonInclude]
    public bool IsAlive = true;
    ...
}
```

Теперь в результате запуска кода любым способом все открытые свойства и поля записываются в файл. Однако, заглянув содержимое файла, вы увидите, что данные JSON были записаны в *минифицированном* виде, т.е. в формате, в котором все незначащие пробельные символы и разрывы строк удаляются. Формат является стандартным во многом из-за широкого использования JSON для служб REST и уменьшения размера пакета данных при передаче информации между службами по HTTP/HTTPS.

На заметку! Поля для сериализации JSON обрабатываются точно так же, как для десериализации JSON. Если вы выбирайте вариант включения полей при сериализации JSON, то также должны делать это при десериализации JSON.

Понятный для человека вывод данных JSON

В дополнение к варианту с включением открытых полей экземпляра класса JsonSerializer можно проинструктировать о необходимости записи данных JSON с отступами (для удобства чтения человеком). Модифицируйте свой метод, как показано ниже:

```
static void SaveAsJsonFormat<T>(T objGraph, string fileName)
{
    var options = new JsonSerializerOptions
    {
        IncludeFields = true,
        WriteIndented = true
    };
    File.WriteAllText(fileName,
        System.Text.Json.JsonSerializer.Serialize(objGraph, options));
}
```

Заглянув в файл CarData.json, вы заметите, что вывод стал гораздо более читабельным:

```
{
    "CanFly": true,
    "CanSubmerge": false,
    "TheRadio": {
```

```

    "HasTweeters": true,
    "HasSubWoofers": false,
    "StationPresets": [
        89.3,
        105.1,
        97.1
    ],
    "RadioId": "XF-552RR6"
},
"IsHatchBack": false
}

```

Именование элементов JSON в стиле Pascal или в “верблюжьем” стиле

Стиль Pascal представляет собой формат, в котором первый символ и каждая важная часть имени начинается с символа в верхнем регистре. В предыдущем листинге данных JSON примером стиля Pascal служит CanSubmerge. В “верблюжьем” стиле, с другой стороны, для первого символа применяется нижний регистр, а все важные части имени начинаются с символа в верхнем регистре. Версия предыдущего примера в “верблюжьем” стиле выглядит как canSubmerge.

Почему это важно? Дело в том, что большинство популярных языков (в том числе C#) чувствительно к регистру. Таким образом, CanSubmerge и canSubmerge — два разных элемента. Повсюду в книге вы видели, что общепринятым стандартом для именования открытых конструкций в C# (классов, открытых свойств, функций и т.д.) является использование стиля Pascal. Тем не менее, в большинстве фреймворков JavaScript задействован “верблюжий” стиль. В итоге могут возникать проблемы при использовании .NET и C# для взаимодействия с другими системами, например, в случае передачи данных JSON туда и обратно между службами REST.

К счастью, JsonSerializer допускает настройку для поддержки большинства ситуаций, в том числе отличий в стилях именования. Если политика именования не указана, то JsonSerializer при сериализации и десериализации JSON будет применять стиль Pascal. Чтобы заставить процесс сериализации использовать “верблюжий” стиль, модифицируйте параметры, как показано ниже:

```

static void SaveAsJsonFormat<T>(T objGraph, string fileName)
{
    JsonSerializerOptions options = new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
        IncludeFields = true,
        WriteIndented = true,
    };
    File.WriteAllText(fileName,
        System.Text.Json.JsonSerializer.Serialize(objGraph, options));
}

```

Теперь выпускаемые данные JSON будут представлены в “верблюжьем” стиле:

```
{
    "canFly": true,
    "canSubmerge": false,
    "theRadio": {
        "hasTweeters": true,
        "hasSubWoofers": false,
    }
}
```

```

"stationPresets": [
    89.3,
    105.1,
    97.1
],
"radioId": "XF-552RR6"
},
"isHatchBack": false
}

```

При чтении данных JSON в коде C# по умолчанию поддерживается чувствительность к регистру символов. Политика именования соответствует настройке `PropertyNamingPolicy`, применяемой во время десериализации. Если ничего не установлено, тогда используется стандартный стиль `Pascal`. Установка `PropertyNamingPolicy` в `CamelCase` свидетельствует об ожидании того, что все входящие данные JSON должны быть представлены в “верблюжьем” стиле. Если политики именования не совпадают, то процесс десериализации (рассматриваемый далее) потерпит неудачу.

При десериализации JSON существует третий вариант — нейтральность к политике именования. Установка параметра `PropertyNameCaseInsensitive` в `true` приводит к тому, что `canSubmerge` и `CanSubmerge` будут десериализироваться. Вот код установки этого параметра:

```

JsonSerializerOptions options = new()
{
    PropertyNameCaseInsensitive = true,
    IncludeFields = true
};

```

Обработка чисел с помощью `JsonSerializer`

Стандартным режимом обработки чисел является `Strict`, который предусматривает, что числа будут сериализоваться как числа (без кавычек) и сериализироваться как числа (без кавычек). В классе `JsonSerializerOptions` имеется свойство `NumberHandling`, которое управляет чтением и записью чисел. В табл. 20.14 перечислены значения, доступные в перечислении `JsonNumberHandling`.

Таблица 20.14. Значения перечисления `JsonNumberHandling`

Значение перечисления	Описание
<code>Strict(0)</code>	Числа читаются как числа и записываются как числа. Кавычки не разрешены и они не генерируются
<code>AllowReadingFromString(1)</code>	Числа читаются из числовых или строковых лексем
<code>WriteAsString(2)</code>	Числа записываются как строки JSON (в кавычках)
<code>AllowNamedFloatingPointLiterals(4)</code>	Могут читаться строковые лексемы <code>Nan</code> , <code>Infinity</code> и <code>-Infinity</code> , а значения с плавающей точкой одинарной и двойной точности будут записываться в виде соответствующих строковых представлений JSON

Перечисление `JsonNumberHandling` имеет атрибут `flags`, который делает возможным побитовое сочетание его значений. Например, если вы хотите читать строки (и числа) и записывать числа в виде строк, тогда применяйте следующую настройку:

58 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
JsonSerializerOptions options = new()
{
    ...
    NumberHandling = JsonNumberHandling.AllowReadingFromString
        & JsonNumberHandling.WriteAsString
};
```

При таком изменении данные JSON, созданные для класса Car, будут выглядеть так:

```
{
    "canFly": true,
    "canSubmerge": false,
    "theRadio": {
        "hasTweeters": true,
        "hasSubWoofers": false,
        "stationPresets": [
            "89.3",
            "105.1",
            "97.1"
        ],
        "radioId": "XF-552RR6"
    },
    "isHatchBack": false
}
```

Потенциальные проблемы, связанные с производительностью, при использовании JsonSerializerOption

В случае применения класса JsonSerializerOption лучше всего создать единственный экземпляр и многократно использовать его повсюду в приложении. С учетом сказанного модифицируйте операторы верхнего уровня и методы, относящиеся к JSON, как показано ниже:

```
JsonSerializerOptions options = new()
{
    PropertyNameCaseInsensitive = true,
   PropertyNamePolicy = JsonNamingPolicy.CamelCase,
    IncludeFields = true,
    WriteIndented = true,
    NumberHandling = JsonNumberHandling.AllowReadingFromString
        | JsonNumberHandling.WriteAsString
};

SaveAsJsonFormat(options, jbc, "CarData.json");
Console.WriteLine("=> Saved car in JSON format!");

SaveAsJsonFormat(options, p, "PersonData.json");
Console.WriteLine("=> Saved person in JSON format!");

static void SaveAsJsonFormat<T>(JsonSerializerOptions options,
                                T objGraph, string fileName)
=> File.WriteAllText(fileName,
    System.Text.Json.JsonSerializer.Serialize(objGraph, options));
```

Стандартные настройки свойств JsonSerializer для веб-приложений

При построении веб-приложений вы можете применять специализированный конструктор для установки следующих свойств:

```
PropertyNameCaseInsensitive = true,
PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
NumberHandling = JsonNumberHandling.AllowReadingFromString
```

Вы по-прежнему можете устанавливать дополнительные свойства через инициализацию объектов, например:

```
JsonSerializerOptions options = new(JsonSerializerDefaults.Web)
{
    WriteIndented = true
};
```

Сериализация коллекций объектов

Сериализация коллекций объектов в данные JSON выполняется аналогично сериализации одиночного объекта. Поместите приведенную далее локальную функцию в конец операторов верхнего уровня:

```
static void SaveListOfCarsAsJson(JsonSerializerOptions options,
                                  string fileName)
{
    // Сохранить список List<T> объектов JamesBondCar.
    List<JamesBondCar> myCars = new()
    {
        new JamesBondCar { CanFly = true, CanSubmerge = true },
        new JamesBondCar { CanFly = true, CanSubmerge = false },
        new JamesBondCar { CanFly = false, CanSubmerge = true },
        new JamesBondCar { CanFly = false, CanSubmerge = false },
    };
    File.WriteAllText(fileName,
        System.Text.Json.JsonSerializer.Serialize(myCars, options));
    Console.WriteLine("=> Saved list of cars!");
}
```

В заключение добавьте следующую строку, чтобы задействовать новую функцию:

```
SaveListOfCarsAsJson(options, "CarCollection.json");
```

Десериализация объектов и коллекций объектов

Как и десериализация XML, десериализация JSON является противоположностью сериализации. Показанная ниже функция будет десериализовать данные JSON в тип, заданный при вызове обобщенной версии метода:

```
static T ReadAsJsonFormat<T>(JsonSerializerOptions options,
                               string fileName) =>
    System.Text.Json.JsonSerializer.Deserialize<T>(
        File.ReadAllText(fileName), options);
```

Добавьте к операторам верхнего уровня следующий код для восстановления объектов (или списка объектов) из данных JSON:

```
JamesBondCar savedJsonCar =
    ReadAsJsonFormat<JamesBondCar>(options, "CarData.json");
Console.WriteLine("Read Car: {0}", savedJsonCar.ToString());
List<JamesBondCar> savedJsonCars =
    ReadAsJsonFormat<List<JamesBondCar>>(options,
                                                "CarCollection.json");
Console.WriteLine("Read Car: {0}", savedJsonCar.ToString());
```

Резюме

Глава начиналась с демонстрации использования типов `Directory` (`DirectoryInfo`) и `File` (`FileInfo`). Вы узнали, что эти классы позволяют манипулировать физическими файлами и каталогами на жестком диске. Затем вы ознакомились с несколькими классами, производными от абстрактного класса `Stream`. Поскольку производные от `Stream` типы оперируют с низкоуровневым потоком байтов, пространство имен `System.IO` предлагает многочисленные типы средств чтения/записи (например, `StreamWriter`, `StringWriter` и `BinaryWriter`), которые упрощают процесс. Попутно вы взглянули на функциональность типа `DriveType`, научились наблюдать за файлами с применением типа `FileSystemWatcher` и выяснили, каким образом взаимодействовать с потоками в асинхронной манере.

В главе также рассматривались службы сериализации объектов. Вы видели, что платформа .NET Core использует граф объектов, чтобы учесть полный набор связанных объектов, которые должны сохраняться в потоке. В заключение вы поработали с сериализацией и десериализацией XML и JSON.

ГЛАВА 21

Доступ к данным с помощью ADO.NET

Внутри платформы .NET Core определено несколько пространств имен, которые позволяют взаимодействовать с реляционными базами данных. Все вместе эти пространства имен известны как ADO.NET. В настоящей главе вы сначала ознакомитесь с общей ролью инфраструктуры ADO.NET, а также основными типами и пространствами имен, после чего будет обсуждаться тема поставщиков данных ADO.NET. Платформа .NET Core поддерживает многочисленные поставщики данных (как являющиеся частью .NET Core, так и доступные от независимых разработчиков), каждый из которых оптимизирован для взаимодействия с конкретной системой управления базами данных (СУБД), например, Microsoft SQL Server, Oracle, MySQL и т.д.

Освоив общую функциональность, предлагаемую различными поставщиками данных, вы узнаете о паттерне фабрики поставщиков данных. Вы увидите, что с использованием типов из пространства имен `System.Data` (включая `System.Data.Common`, а также специфичные для поставщиков данных пространства имен вроде `Microsoft.Data.SqlClient`, `System.Data.Odbc` и доступное только в Windows пространство имен `System.Data.OleDb`) можно строить единственную кодовую базу, которая способна динамически выбирать поставщик данных без необходимости в повторной компиляции или развертывании кодовой базы приложения.

Далее вы научитесь работать напрямую с поставщиком баз данных `SQL Server`, создавая и открывая подключения для извлечения данных и затем вставляя, обновляя и удаляя данные, и ознакомитесь с темой транзакций базы данных. Наконец, вы запустите средство массового копирования `SQL Server` с применением ADO.NET для загрузки списка записей внутрь базы данных.

На заметку! Внимание в настоящей главе сконцентрировано на низкоуровневой инфраструктуре ADO.NET. Начиная с главы 22, будет раскрываться инфраструктура объектно-реляционного отображения (object-relational mapping — ORM) производства Microsoft под названием Entity Framework (EF) Core. Поскольку инфраструктура EF Core для доступа к данным внутренне использует ADO.NET, хорошее понимание принципов работы ADO.NET жизненно важно при поиске и устранении проблем при доступе к данным. Кроме того, существуют задачи, решить которые с помощью EF Core не удастся (такие как выполнение массового копирования данных в SQL), и для их решения требуются знания ADO.NET.

Сравнение ADO.NET и ADO

Если у вас есть опыт работы с предшествующей моделью доступа к данным на основе COM от Microsoft (Active Data Objects — ADO) и вы только начинаете использовать платформу .NET Core, то имейте в виду, что инфраструктура ADO.NET имеет мало общего с ADO помимо наличия в своем названии букв “A”, “D” и “O”. В то время как определенная взаимосвязь между двумя системами действительно существует (скажем, в обеих присутствует концепция объектов подключений и объектов команд), некоторые знакомые по ADO типы (например, Recordset) больше не доступны. Вдобавок вы обнаружите много новых типов, которые не имеют прямых эквивалентов в классической технологии ADO (скажем, адаптер данных).

Поставщики данных ADO.NET

В ADO.NET нет единого набора типов, которые взаимодействовали бы с множеством СУБД. Взамен ADO.NET поддерживает многочисленные поставщики данных, каждый из которых оптимизирован для взаимодействия со специфичной СУБД. Первое преимущество такого подхода в том, что вы можете запрограммировать специализированный поставщик данных для доступа к любым уникальным средствам отдельной СУБД. Второе преимущество связано с тем, что поставщик данных может подключаться непосредственно к механизму интересующей СУБД без какого-либо промежуточного уровня отображения.

Выражаясь просто, поставщик данных — это набор типов, определенных в отдельном пространстве имен, которым известно, как взаимодействовать с конкретным источником данных. Безотносительно к тому, какой поставщик данных применяется, каждый из них определяет набор классов, представляющих основную функциональность. В табл. 21.1 описаны распространенные основные базовые классы и ключевые интерфейсы, которые они реализуют.

Хотя конкретные имена основных классов будут отличаться между поставщиками данных (например, SqlConnection в сравнении с OdbcConnection), все они являются производными от того же самого базового класса (DbConnection в случае объектов подключения), реализующего идентичные интерфейсы (вроде IDbConnection). С учетом сказанного вполне корректно предположить, что после освоения работы с одним поставщиком данных остальные поставщики покажутся довольно простыми.

На заметку! Когда речь идет об объекте подключения в ADO.NET, то на самом деле имеется в виду специфичный тип, производный от DbConnection; не существует класса с буквальным именем “Connection”. Та же идея остается справедливой в отношении объекта команды, объекта адаптера данных и т.д. По соглашению об именовании объекты в конкретном поставщике данных снабжаются префиксом в форме названия связанной СУБД (например, SqlConnection, SqlCommand и SqlDataReader).

На рис. 21.1 иллюстрируется место поставщиков данных в инфраструктуре ADO.NET. Клиентская сборка может быть приложением .NET Core любого типа: консольной программой, приложением Windows Forms, приложением WPF, веб-страницей ASP.NET Core, библиотекой кода .NET Core и т.д.

Кроме типов, показанных на рис. 21.1, поставщики данных будут предоставлять и другие типы; однако эти основные объекты определяют общие характеристики для всех поставщиков данных.

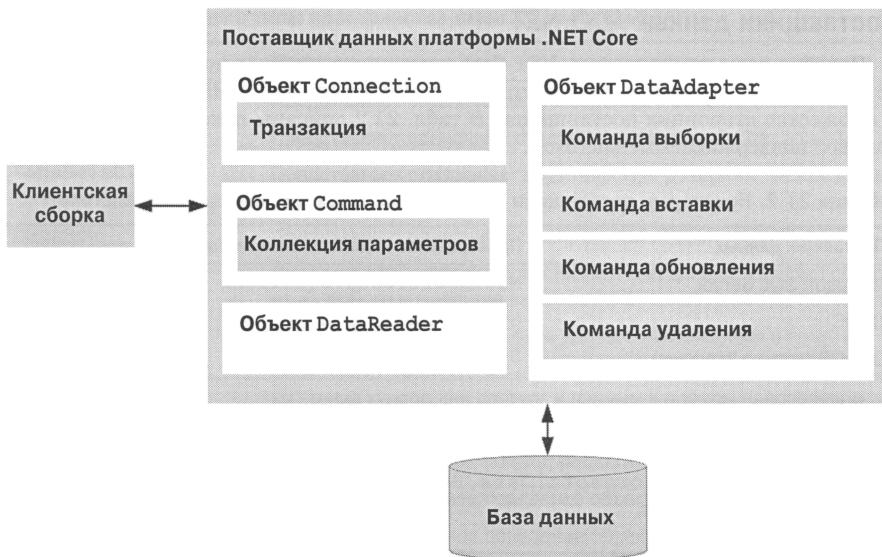


Рис. 21.1. Поставщики данных ADO.NET предоставляют доступ к конкретным СУБД

Таблица 21.1. Основные типы поставщиков данных ADO.NET

Базовый класс	Реализуемые интерфейсы	Описание
DbConnection	IDbConnection	Предоставляет возможность подключения и отключения от хранилища данных. Объекты подключения также обеспечивают доступ к связанному объекту транзакции
DbCommand	IDbCommand	Представляет запрос SQL или хранимую процедуру. Объекты команд также предлагают доступ к объекту чтения данных поставщика
DbDataReader	IDataReader, IDataRecord	Обеспечивает доступ к данным в направлении только вперед, допускающий только чтение, с использованием курсора на стороне сервера
DbDataAdapter	IDataAdapter, IDbDataAdapter	Передает объекты DataSet между вызывающим кодом и хранилищем данных. Адаптеры данных содержат объект подключения и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации из хранилища данных
DbParameter	IDataParameter, IDbDataParameter	Представляет именованный параметр внутри параметризованного запроса
DbTransaction	IDbTransaction	Инкапсулирует транзакцию базы данных

Поставщики данных ADO.NET

Подобно всем компонентам .NET Core поставщики данных поступают в виде пакетов NuGet. В их число входят поставщики, поддерживаемые Microsoft, но доступно и множество сторонних поставщиков. В табл. 21.2 описаны некоторые поставщики данных, поддерживаемые Microsoft.

Таблица 21.2. Некоторые поставщики данных ADO.NET, поддерживаемые Microsoft

Поставщик данных	Пространство имен/имя пакета NuGet
Microsoft SQL Server	Microsoft.Data.SqlClient
ODBC	System.Data.Odbc
OLE DB (только Windows)	System.Data.OleDb

Поставщик данных Microsoft SQL Server предлагает прямой доступ к хранилищам данных Microsoft SQL Server — и только к ним (включая SQL Azure). Пространство имен Microsoft.Data.SqlClient содержит типы, используемые поставщиком SQL Server.

На заметку! Хотя System.Data.SqlClient по-прежнему поддерживается, все усилия по разработке средств для взаимодействия с SQL Server (и с SQL Azure) сосредоточены на новой библиотеке поставщика Microsoft.Data.SqlClient.

Поставщик ODBC (System.Data.Odbc) обеспечивает доступ к подключениям ODBC. Типы ODBC, определенные в пространстве имен System.Data.Odbc, обычно полезны, только если требуется взаимодействие с СУБД, для которой отсутствует специальный поставщик данных .NET Core. Причина в том, что ODBC является широко распространенной моделью, которая предоставляет доступ к нескольким хранилищам данных.

Поставщик данных OLE DB, который состоит из типов, определенных в пространстве имен System.Data.OleDb, позволяет получать доступ к данным в любом хранилище данных, поддерживающем классический протокол OLE DB на основе COM. Из-за зависимости от COM этот поставщик будет работать только в среде Windows и считается устаревшим в межплатформенном мире .NET Core.

Типы из пространства имен System.Data

Из всех пространств имен, относящихся к ADO.NET, System.Data является “наименьшим общим знаменателем”. Оно содержит типы, которые совместно используются всеми поставщиками данных ADO.NET независимо от лежащего в основе хранилища данных. В дополнение к нескольким исключениям, связанным с базами данных (например, `NoNullAllowedException`, `RowNotInTableException` и `MissingPrimaryKeyException`), пространство имен System.Data содержит типы, которые представляют разнообразные примитивы баз данных (вроде таблиц, строк, столбцов и ограничений), а также общие интерфейсы, реализуемые классами поставщиков данных. В табл. 21.3 описаны основные типы, о которых следует знать.

Таблица 21.3. Основные типы пространства имен System.Data

Тип	Описание
Constraint	Представляет ограничение для заданного объекта DataColumn
DataTableColumn	Представляет одиночный столбец внутри объекта DataTable
DataRelation	Представляет отношение "родительский–дочерний" между двумя объектами DataTable
DataRow	Представляет одиночную строку внутри объекта DataTable
DataSet	Представляет находящийся в памяти кеш данных, который состоит из любого количества взаимосвязанных объектов DataTable
DataTable	Представляет табличный блок данных, находящийся в памяти
DataTableReader	Позволяет трактовать объект DataTable как допускающий только чтение курсор для доступа к данным в прямом направлении
DataView	Определяет настраиваемое представление DataTable для сортировки, фильтрации, поиска, редактирования и навигации
IDataAdapter	Определяет основное поведение объекта адаптера данных
IDataParameter	Определяет основное поведение объекта параметра
IDataReader	Определяет основное поведение объекта чтения данных
IDbCommand	Определяет основное поведение объекта команды
IDbDataAdapter	Расширяет интерфейс IDataAdapter для обеспечения объекта адаптера данных дополнительной функциональностью
IDbTransaction	Определяет основное поведение объекта транзакции

Следующей задачей будет исследование основных интерфейсов System.Data на высоком уровне, что поможет лучше понять общую функциональность, предлагаемую любым поставщиком данных. В ходе чтения настоящей главы вы также ознакомитесь с конкретными деталями, а пока лучше сосредоточить внимание на общем поведении каждого типа интерфейса.

Роль интерфейса IDbConnection

Интерфейс IDbConnection реализован объектом подключения поставщика данных. В нем определен набор членов, применяемых для конфигурирования подключения к специальному хранилищу данных. Он также позволяет получить объект транзакции поставщика данных. Вот формальное определение IDbConnection:

```
public interface IDbConnection : IDisposable
{
    string ConnectionString { get; set; }
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }

    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
```

```

    IDbCommand CreateCommand();
    void Open();
    void Dispose();
}

```

Роль интерфейса IDbTransaction

Перегруженный метод `BeginTransaction()`, определенный в интерфейсе `IDbConnection`, предоставляет доступ к *объекту транзакции* поставщика. Члены, определенные интерфейсом `IDbTransaction`, позволяют программно взаимодействовать с транзакционным сеансом и лежащим в основе хранилищем данных:

```

public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }

    void Commit();
    void Rollback();
    void Dispose();
}

```

Роль интерфейса IDbCommand

Интерфейс `IDbCommand` будет реализован *объектом команды* поставщика данных. Подобно другим объектным моделям доступа к данным объекты команд позволяют программно манипулировать операторами SQL, хранимыми процедурами и параметризованными запросами. Объекты команд также обеспечивают доступ к типу чтения данных поставщика данных посредством перегруженного метода `ExecuteReader()`:

```

public interface IDbCommand : IDisposable
{
    string CommandText { get; set; }
    int CommandTimeout { get; set; }
    CommandType CommandType { get; set; }
    IDbConnection Connection { get; set; }
    IDbTransaction Transaction { get; set; }
    IDataParameterCollection Parameters { get; }
    UpdateRowSource UpdatedRowSource { get; set; }

    void Prepare();
    void Cancel();
    IDbDataParameter CreateParameter();
    int ExecuteNonQuery();
    IDataReader ExecuteReader();
    IDataReader ExecuteReader(CommandBehavior behavior);
    object ExecuteScalar();
    void Dispose();
}

```

Роль интерфейсов IDbDataParameter и IDataParameter

Обратите внимание, что свойство `Parameters` интерфейса `IDbCommand` возвращает строго типизированную коллекцию, реализующую интерфейс `IDataParameterCollection`, который предоставляет доступ к набору классов, совместимых с `IDbDataParameter` (например, объектам параметров):

```
public interface IDbDataParameter : IDataParameter
{
    // Плюс члены интерфейса IDataParameter.
    byte Precision { get; set; }
    byte Scale { get; set; }
    int Size { get; set; }
}
```

Интерфейс `IDbDataParameter` расширяет `IDataParameter` с целью обеспечения дополнительных линий поведения:

```
public interface IDataParameter
{
    DbType DbType { get; set; }
    ParameterDirection Direction { get; set; }
    bool IsNullable { get; }
    string ParameterName { get; set; }
    string SourceColumn { get; set; }
    DataRowVersion SourceVersion { get; set; }
    object Value { get; set; }
}
```

Вы увидите, что функциональность интерфейсов `IDbDataParameter` и `IDataParameter` позволяет представлять параметры внутри команды SQL (включая хранимые процедуры) с помощью специфических объектов параметров ADO.NET вместо жестко закодированных строковых литералов.

Роль интерфейсов `IDbDataAdapter` и `IDataAdapter`

Адаптеры данных используются для помещения объектов `DataSet` в хранилище данных и извлечения их из него. Интерфейс `IDbDataAdapter` определяет следующий набор свойств, которые можно применять для поддержки операторов SQL, выполняющих связанные операции выборки, вставки, обновления и удаления:

```
public interface IDbDataAdapter : IDataAdapter
{
    // Плюс члены интерфейса IDataAdapter.
    IDbCommand SelectCommand { get; set; }
    IDbCommand InsertCommand { get; set; }
    IDbCommand UpdateCommand { get; set; }
    IDbCommand DeleteCommand { get; set; }
}
```

В дополнение к показанным четырем свойствам адаптер данных ADO.NET также получает линии поведения, определенные базовым интерфейсом, т.е. `IDataAdapter`. Интерфейс `IDataAdapter` определяет ключевую функцию типа адаптера данных: способность передавать объекты `DataSet` между вызывающим кодом и внутренним хранилищем данных, используя методы `Fill()` и `Update()`. Кроме того, интерфейс `IDataAdapter` позволяет с помощью свойства `TableMappings` сопоставлять имена столбцов базы данных с более дружественными к пользователю отображаемыми именами:

```
public interface IDataAdapter
{
    MissingMappingAction MissingMappingAction { get; set; }
    MissingSchemaAction MissingSchemaAction { get; set; }
    ITableMappingCollection TableMappings { get; }
```

```

DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
int Fill(DataSet dataSet);
IDataParameter[] GetFillParameters();
int Update(DataSet dataSet);
}

```

Роль интерфейсов **IDataReader** и **IDataRecord**

Следующим основным интерфейсом является **IDataReader**, который представляет общие линии поведения, поддерживаемые отдельно взятым объектом чтения данных. После получения от поставщика данных ADO.NET объекта совместимого с **IDataReader** типа можно выполнять проход по результирующему набору в прямом направлении с поддержкой только чтения.

```

public interface IDataReader : IDisposable, IDataRecord
{
    // Плюс члены интерфейса IDataRecord.
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }

    void Close();
    DataTable GetSchemaTable();
    bool NextResult();
    bool Read();
    Dispose();
}

```

Наконец, интерфейс **IDataReader** расширяет **IDataRecord**. В интерфейсе **IDataRecord** определено много членов, которые позволяют извлекать из потока строго типизированное значение, а не приводить к нужному типу экземпляр **System.Object**, полученный из перегруженного метода индексатора объекта чтения данных. Вот определение интерфейса **IDataRecord**:

```

public interface IDataRecord
{
    int FieldCount { get; }
    object this[ int i ] { get; }
    object this[ string name ] { get; }
    bool GetBoolean(int i);
    byte GetByte(int i);
    long GetBytes(int i, long fieldOffset, byte[] buffer,
        int bufferoffset, int length);
    char GetChar(int i);
    long GetChars(int i, long fieldoffset, char[] buffer,
        int bufferoffset, int length);
    IDataReader GetData(int i);
    string GetDataTypeName(int i);
    DateTime GetDateTime(int i);
    Decimal GetDecimal(int i);
    double GetDouble(int i);
    Type GetFieldType(int i);
    float GetFloat(int i);
    Guid GetGuid(int i);
    short GetInt16(int i);
}

```

```

int GetInt32(int i);
long GetInt64(int i);
string GetName(int i);
int GetOrdinal(string name);
string GetString(int i);
object GetValue(int i);
int GetValues(object[] values);
bool IsDBNull(int i);
}

```

На заметку! Метод `IDataReader.IsDBNull()` можно применять для программного выяснения, установлено ли указанное поле в `null`, прежде чем пытаться получить значение из объекта чтения данных (во избежание генерации исключения во время выполнения). Также вспомните, что язык C# поддерживает типы данных, допускающие `null` (см. главу 4), идеально подходящие для взаимодействия со столбцами, которые могут иметь значение `null` в таблице базы данных.

Абстрагирование поставщиков данных с использованием интерфейсов

К настоящему моменту вы должны лучше понимать общую функциональность, присущую всем поставщикам данных .NET Core. Вспомните, что хотя точные имена реализуемых типов будут отличаться между поставщиками данных, в коде такие типы применяются в схожей манере — в том и заключается преимущество полиморфизма на основе интерфейсов. Скажем, если определить метод, который принимает параметр `IDbConnection`, то ему можно передавать любой объект подключения ADO.NET:

```

public static void OpenConnection(IDbConnection connection)
{
    // Открыть входное подключение для вызывающего кода.
    connection.Open();
}

```

На заметку! Использовать интерфейсы вовсе не обязательно; аналогичного уровня абстракции можно достичь путем применения абстрактных базовых классов (таких как `DbConnection`) в качестве параметров или возвращаемых значений. Однако использование интерфейсов вместо базовых классов является общепринятой практикой.

То же самое справедливо для возвращаемых значений. Создайте новый проект консольного приложения .NET Core по имени `MyConnectionFactory`. Добавьте в проект перечисленные ниже пакеты NuGet (пакет `OleDb` действителен только в Windows):

```

Microsoft.Data.SqlClient
System.Data.Common
System.Data.Odbc
System.Data.OleDb

```

Далее добавьте в проект новый файл по имени `DataProviderEnum.cs` со следующим кодом:

70 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
namespace MyConnectionFactory
{
    // Пакет OleDb предназначен только для Windows и в .NET Core
    // не поддерживается.
    enum DataProviderEnum
    {
       SqlServer,
#if PC
        OleDb,
#endif
        Odbc,
        None
    }
}
```

Если на своей машине обработки вы работаете в среде Windows, тогда модифицируйте файл проекта, чтобы определить символ условной компиляции PC:

```
<PropertyGroup>
<DefineConstants>PC</DefineConstants>
</PropertyGroup>
```

В случае использования Visual Studio щелкните правой кнопкой мыши на имени проекта и выберите в контекстном меню пункт Properties (Свойства). В открывшемся диалоговом окне Properties (Свойства) перейдите на вкладку Build (Сборка) и введите нужное значение в поле Conditional compiler symbols (Символы условной компиляции).

Следующий пример кода позволяет выбирать специфический объект подключения на основе значения из специального перечисления. В целях диагностики мы просто выводим лежащий в основе объект подключения с применением служб рефлексии.

```
using System;
using System.Data;
using System.Data.Odbc;
#if PC
    using System.Data.OleDb;
#endif
using Microsoft.Data.SqlClient;
using MyConnectionFactory;

Console.WriteLine("***** Very Simple Connection Factory *****\n");
Setup(DataProviderEnum.SqlServer);
#if PC
    Setup(DataProviderEnum.OleDb); // Не поддерживается в macOS.
#endif
Setup(DataProviderEnum.Odbc);
Setup(DataProviderEnum.None);
Console.ReadLine();

void Setup(DataProviderEnum provider)
{
    // Получить конкретное подключение.
    IDbConnection myConnection = GetConnection(provider);
    Console.WriteLine($"Your connection is a {myConnection?.GetType().Name
?? "unrecognized type"}");
    // Открыть, использовать и закрыть подключение...
}
```

```
// Этот метод возвращает конкретный объект подключения
// на основе значения перечисления DataProvider.
IDbConnection GetConnection(DataProviderEnum dataProvider)
    => dataProvider switch
    {
        DataProviderEnum.SqlServer => new SqlConnection(),
#if PC
        // Не поддерживается в macOS.
        DataProviderEnum.OleDb => new OleDbConnection(),
#endif
        DataProviderEnum.Odbc => new OdbcConnection(),
        _ => null,
    };
}
```

Преимущество работы с общими интерфейсами из пространства имен `System.Data` (или на самом деле с абстрактными базовыми классами из пространства имен `System.Data.Common`) связано с более высокими шансами построить гибкую кодовую базу, которую со временем можно развивать. Например, в настоящий момент вы можете разрабатывать приложение, предназначеннное для Microsoft SQL Server; тем не менее, вполне возможно, что спустя несколько месяцев ваша компания перейдет на другую СУБД. Если вы строите решение с жестко закодированными типами из пространства имен `System.Data.SqlClient`, которые специфичны для Microsoft SQL Server, тогда вполне очевидно, что в случае смены серверной СУБД код придется редактировать, заново компилировать и развертывать.

К настоящему моменту вы написали (довольно простой) код ADO.NET, который позволяет создавать различные типы объектов подключений, специфичные для поставщика. Тем не менее, получение объекта подключения — лишь один аспект работы с ADO.NET. Чтобы построить полезную библиотеку фабрики поставщиков данных, необходимо также учитывать объекты команд, объекты чтения данных, адаптеры данных, объекты транзакций и другие типы, связанные с данными. Создание подобной библиотеки кода не обязательно будет трудным, но все-таки потребует написания значительного объема кода и затрат времени.

Начиная с версии .NET 2.0, такая функциональность встроена прямо в библиотеки базовых классов .NET. В .NET Core эта функциональность была значительно обновлена.

Вскоре мы исследуем упомянутый формальный API-интерфейс, но сначала понадобится создать специальную базу данных для применения в настоящей главе (и во многих последующих главах).

Установка SQL Server и Azure Data Studio

На протяжении оставшегося материала главы мы будем выполнять запросы в отношении простой тестовой базы данных SQL Server по имени `AutoLot`. В продолжение автомобильной темы, затрагиваемой повсеместно в книге, база данных `AutoLot` будет содержать пять взаимосвязанных таблиц (`Inventory`, `Makes`, `Orders`, `Customers` и `CreditRisks`), которые хранят различные данные о заказах гипотетической компании по продаже автомобилей. Прежде чем погрузиться в детали, связанные с базой данных, вы должны установить SQL Server и IDE-среду SQL Server.

На заметку! Если ваша машина для разработки функционирует под управлением Windows и вы установили Visual Studio 2019, то уже имеете установленный экземпляр SQL Server Express (под названием localdb), который можно использовать для всех примеров в настоящей книге. В случае согласия работать с указанной версией можете сразу переходить в раздел "Установка IDE-среды SQL Server".

Установка SQL Server

В текущей главе и многих оставшихся главах книги вам будет нужен доступ к экземпляру SQL Server. Если вы применяете машину разработки, на которой установлена ОС, отличающаяся от Windows, и у вас нет доступного внешнего экземпляра SQL Server, или вы решили не использовать внешний экземпляр SQL Server, то можете запустить SQL Server внутри контейнера Docker на рабочей станции Mac или Linux. Контейнер Docker функционирует и на машинах Windows, поэтому вы можете выполнять примеры в книге с применением Docker независимо от выбранной ОС.

Установка SQL Server в контейнер Docker

В случае использования машины разработки, основанной не на Windows, и отсутствии доступного для примеров экземпляра SQL Server вы можете запустить SQL Server внутри контейнера Docker на рабочей станции Mac или Linux. Контейнер Docker работает также на машинах Windows, поэтому вы можете выполнять примеры в книге с применением Docker независимо от выбранной ОС.

На заметку! Контейнеризация является крупной темой, и в этой книге просто нет места, чтобы углубиться в подробности контейнеров или Docker. Книга охватывает ровно столько, чтобы вы могли проработать примеры.

Docker Desktop можно загрузить по ссылке www.docker.com/get-started. Загрузите и установите подходящую версию (Windows, Mac, Linux) для своей рабочей станции (вам понадобится учетная запись DockerHub). Удостоверьтесь, что при выдаче запроса выбраны контейнеры Linux.

На заметку! Выбранный вариант контейнера (Windows или Linux) — это ОС, функционирующая внутри контейнера, а не ОС, установленная на рабочей станции.

Получение образа и запуск SQL Server 2019

Контейнеры основаны на образах, а каждый образ представляет собой многоуровневый набор, из которого образован финальный продукт. Чтобы получить образ, необходимый для запуска SQL Server 2019 в контейнере, откройте окно командной строки и введите следующую команду:

```
docker pull mcr.microsoft.com/mssql/server:2019-latest
```

После загрузки образа на машину вам понадобится запустить SQL Server, для чего ввести следующую команду (целиком в одной строке):

```
docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=P@ssw0rd"
-p 5433:1433 --name AutoLot
-d mcr.microsoft.com/mssql/server:2019-latest
```

Предыдущая команда принимает лицензионное соглашение конечного пользователя, устанавливает пароль (в реальности будет использоваться строгий пароль), устанавливает отображение портов (порт 5433 на вашей машине отображается на стандартный порт для SQL Server в контейнере (1433)), указывает имя контейнера (AutoLot) и, наконец, информирует Docker о том, что должен применяться ранее загруженный образ.

На заметку! Это не те настройки, которые вы захотите использовать в реальной разработке.

Информация о том, как изменить пароль системного администратора, и другие сведения доступны по ссылке <https://docs.microsoft.com/ru-ru/sql/linux/quickstart-install-connect-docker?view=sql-server-ver15&pivots=csl-bash>.

Чтобы убедиться в том, что Docker функционирует, введите в окне командной строки команду docker ps -a. Вы увидите вывод наподобие показанного ниже (для краткости некоторые колонки опущены):

```
C:\Users\japik>docker ps -a
CONTAINER ID IMAGE STATUS
PORTS NAMES
347475cfb823 mcr.microsoft.com/mssql/server:2019-latest Up 6 minutes
0.0.0.0:5433->1433/tcp AutoLot
```

Чтобы остановить контейнер, введите docker stop 34747, где цифры 34747 представляют собой первые пять символов идентификатора контейнера. Чтобы перезапустить контейнер, введите docker start 34747, не забыв обновить команду соответствующим началом идентификатора вашего контейнера.

На заметку! Вы также можете использовать с командами Docker CLI имя контейнера (AutoLot в этом примере), скажем, docker start AutoLot. Имейте в виду, что независимо от ОС команды Docker чувствительны к регистру символов.

Если вы желаете поработать с инструментальной панелью Docker, щелкните правой кнопкой мыши на значке Docker (в системном лотке) и выберите в контекстном меню пункт Dashboard (Инструментальная панель); вы должны увидеть образ, функционирующий на порте 5433. Наведите курсор мыши на имя образа и появятся команды для остановки, запуска и удаления (помимо прочих), как показано на рис. 21.2.



Рис. 21.2. Инструментальная панель Docker

Установка SQL Server 2019

Вместе с Visual Studio 2019 устанавливается специальный экземпляр SQL Server (по имени (localdb)\mssqllocaldb). Если вы решили не использовать SQL Server Express LocalDB (или Docker) и работаете на машине Windows, тогда можете установить SQL Server 2019 Developer Edition. Продукт SQL Server 2019 Developer Edition бесплатен и доступен для загрузки по следующей ссылке:

<https://www.microsoft.com/ru-ru/sql-server/sql-server-downloads>

Имея другую версию экземпляра, вы можете применять ее в этой книге; понадобится лишь надлежащим образом изменить параметры на экране подключения.

Установка IDE-среды SQL Server

Azure Data Studio — это новая IDE-среда для использования с SQL Server. Она является бесплатной и межплатформенной, а потому будет работать под управлением Windows, Mac или Linux. Загрузить ее можно по ссылке:

<https://docs.microsoft.com/ru-ru/sql/azure-data-studio/download-azure-data-studio>

На заметку! Если вы работаете на машине Windows и отдаете предпочтение среде управления SQL Server (SQL Server Management Studio — SSMS), то можете загрузить последнюю версию по ссылке <https://docs.microsoft.com/ru-ru/sql/ssms/download-sql-server-management-studio-ssms>.

Подключение к SQL Server

После установки Azure Data Studio или SSMS настало время подключиться к экземпляру СУБД. В последующих разделах описано подключение к SQL Server в контейнере Docker или LocalDb. Если вы используете другой экземпляр SQL Server, тогда соответствующим образом обновите строку подключения.

Подключение к SQL Server в контейнере Docker

Чтобы подключиться к экземпляру SQL Server, функционирующему в контейнере Docker, сначала убедитесь, что он запущен и работает. Затем щелкните на элементе Create a connection (Создать подключение) в Azure Data Studio (рис. 21.3).

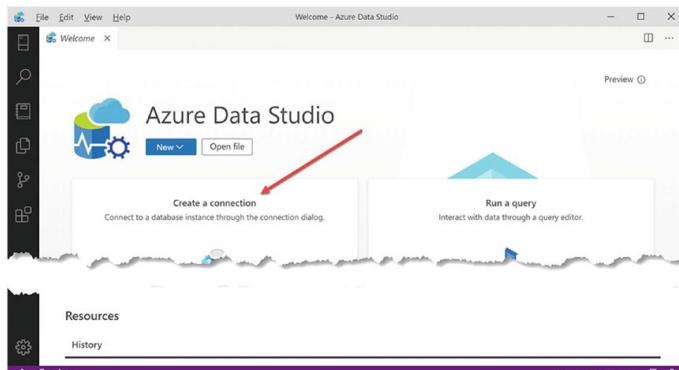


Рис. 21.3. Создание подключения в Azure Data Studio

В диалоговом окне Connection Details (Детали подключения) введите ., 5433 в поле Server (Сервер). Точка задает текущий хост, а ,5433 — это порт, который вы указывали при создании экземпляра SQL Server в контейнере Docker. Введите sa в поле User name (Имя пользователя); пароль остается тем же самым, который вводился при создании экземпляра SQL Server. Имя в поле Name (Имя) является необязательным, но оно позволяет быстро выбирать данное подключение в последующих сессиях Azure Data Studio. Упомянутые параметры подключения показаны на рис. 21.4.

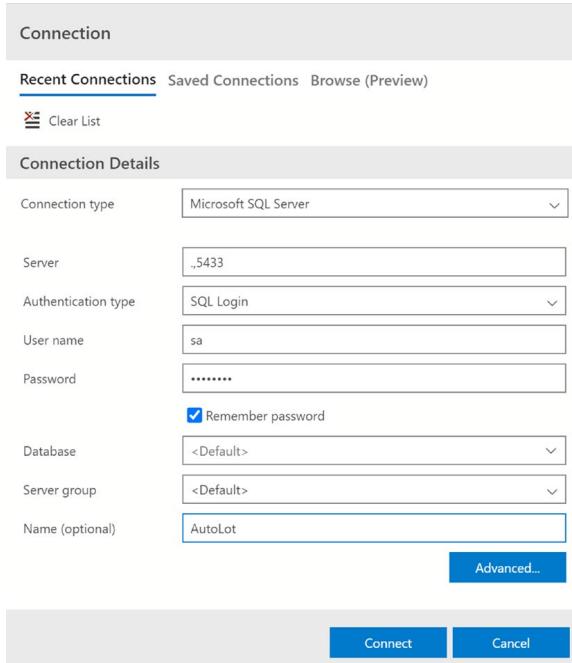


Рис. 21.4. Настройка параметров подключения для экземпляра SQL Server в контейнере Docker

Подключение к SQL Server LocalDb

Чтобы подключиться к версии SQL Server Express LocalDb, установленной вместе с Visual Studio, приведите информацию о подключении в соответствие с показанной на рис. 21.5.

При подключении к LocalDb вы можете использовать аутентификацию Windows, поскольку экземпляр работает на той же машине, что и Azure Data Studio, и в том же контексте безопасности, что и текущий вошедший в систему пользователь.

Подключение к любому другому экземпляру SQL Server

Если вы подключаетесь к любому другому экземпляру SQL Server, тогда соответствующим образом обновите параметры подключения.

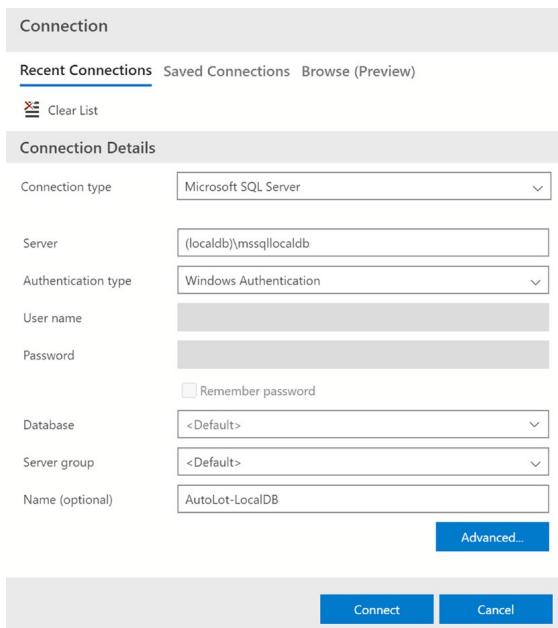


Рис. 21.5. Установка параметров подключения для SQL Server LocalDb

Восстановление базы данных AutoLot из резервной копии

Вместо того чтобы создавать базу данных с нуля, вы можете воспользоваться SSMS или Azure Data Studio для восстановления одной из резервных копий, содержащихся в папке Chapter_21 хранилища GitHub для данной книги. Предлагаются две резервные копии: одна по имени AutoLotWindows.ba_ рассчитана на применение на машине Windows (LocalDb, Windows Server и т.д.) и еще одна по имени AutoLotDocker.ba_ предназначена для использования в контейнере Docker.

На заметку! GitHub по умолчанию игнорирует файлы с расширением `bak`. Прежде чем восстанавливать базу данных, вам придется переименовать расширение `ba_` в `bak`.

Копирование файла резервной копии в имеющийся контейнер

Если вы работаете с SQL Server в контейнере Docker, то сначала должны скопировать файл резервной копии в контейнер. К счастью, Docker CLI предлагает механизм для взаимодействия с файловой системой контейнера. Первым делом создайте новый каталог для резервной копии с помощью следующей команды в окне командной строки на хост-машине:

```
docker exec -it AutoLot mkdir var/opt/mssql/backup
```

Структура пути должна соответствовать ОС контейнера (в данном случае Ubuntu), даже если хост-машина функционирует под управлением Windows. Затем скопируйте резервную копию с применением показанной ниже команды (укажите для местоположения файла AutoLotDocker.bak относительный или абсолютный путь на вашей локальной машине):

```
[Windows]
docker cp .\AutoLotDocker.bak AutoLot:/var/opt/mssql/backup
```

```
[Non-Windows]
docker cp ./AutoLotDocker.bak AutoLot:/var/opt/mssql/backup
```

Обратите внимание, что исходная структура каталогов соответствует хост-машине (в этом примере Windows), тогда как цель выглядит как имя контейнера и затем путь к каталогу (в формате целевой ОС).

Восстановление базы данных с помощью SSMS

Чтобы восстановить базу данных с применением SSMS, щелкните правой кнопкой мыши на узле Databases (Базы данных) в проводнике объектов и выберите в контекстном меню пункт Restore Database (Восстановить базу данных). Укажите вариант Device (Устройство) и щелкните на символе троеточия. Откроется диалоговое окно Select Backup Device (Выбор устройства с резервной копией).

Восстановление базы данных в экземпляре SQL Server (Docker)

Оставив в раскрывающемся списке Backup media type (Тип носителя резервной копии) выбранным вариант File (Файл), щелкните на кнопке Add (Добавить), перейдите к файлу AutoLotDocker.bak в контейнере и щелкните на кнопке OK. Возвратившись в главное диалоговое окно восстановления, щелкните на кнопке OK (рис. 21.6).

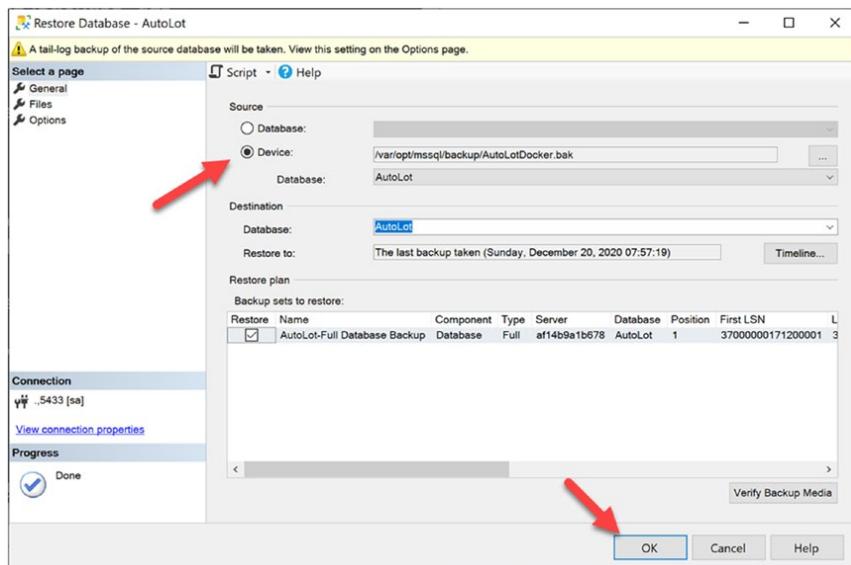


Рис. 21.6. Восстановление базы данных посредством SSMS

Восстановление базы данных в экземпляре SQL Server (Windows)

Оставив в раскрывающемся списке Backup media type выбранным вариантом File, щелкните на кнопке Add, перейдите к файлу AutoLotWindows.bak и щелкните на кнопке OK. Возвратившись в главное диалоговое окно восстановления, щелкните на кнопке OK (рис. 21.7).

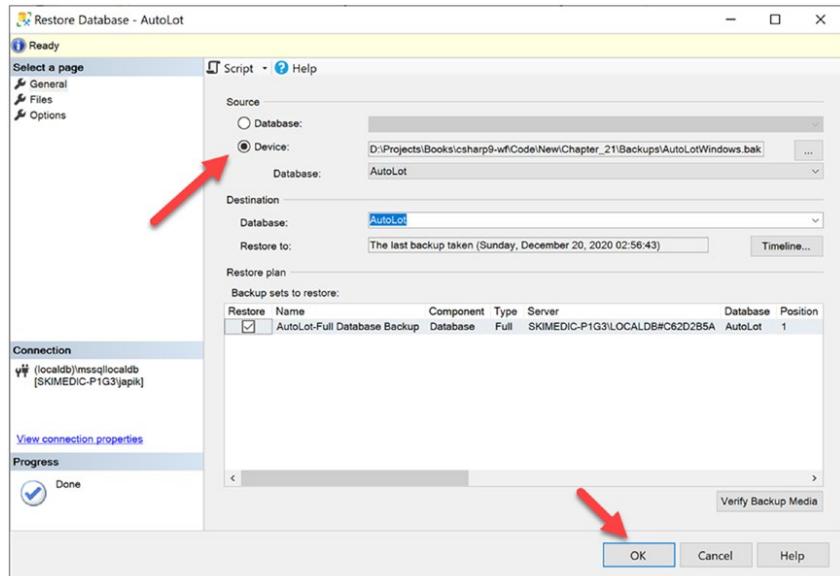


Рис. 21.7. Восстановление базы данных с применением SSMS

Восстановление базы данных с помощью Azure Data Studio

Чтобы восстановить базу данных с использованием Azure Data Studio, выберите в области Tasks (Задачи) вариант Restore (Восстановить). Укажите в раскрывающемся списке Restore from (Восстановить из) вариант Backup file (Файл резервной копии) и затем выберите только что скопированный файл. Целевая база данных и связанные поля заполняются автоматически, как показано на рис. 21.8.

На заметку! Процесс восстановления версии Windows резервной копии посредством Azure Data Studio аналогичен. Понадобится просто скорректировать имя файла и пути.

Создание базы данных AutoLot

Весь этот раздел посвящен созданию базы данных AutoLot с применением Azure Data Studio. Если вы используете SSMS, то можете выполнить описанные здесь шаги, применяя либо приведенные SQL-схемарии, либо инструменты с графическим пользовательским интерфейсом. Если вы восстановили резервную копию, тогда переходите сразу в раздел “Модель фабрики поставщиков данных ADO.NET”.

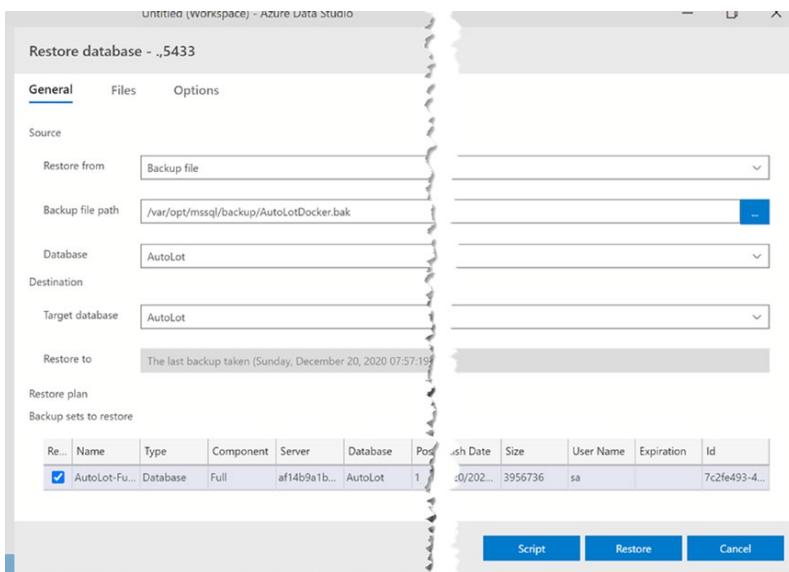


Рис. 21.8. Восстановление базы данных в Docker с использованием Azure Data Studio

На заметку! Все файлы сценариев находятся в подпапке по имени `Scripts` внутри папки `Chapter_21` хранилища GitHub для данной книги.

Создание базы данных

Для создания базы данных `AutoLot` подключитесь к своему серверу баз данных с использованием Azure Data Studio. Откройте окно нового запроса, выбрав пункт меню `File⇒New Query` (Файл⇒Новый запрос) или нажав комбинацию `<Ctrl+N>`, и введите следующие команды SQL:

```
USE [master]
GO
/***** Object: Database [AutoLot50]    Script Date: 12/20/2020 01:48:05 *****/
CREATE DATABASE [AutoLot]
GO
ALTER DATABASE [AutoLot50] SET RECOVERY SIMPLE
GO
```

Кроме изменения режима восстановления на простой команда создает базу данных `AutoLot` с применением стандартных параметров SQL Server. Щелкните на кнопке `Run` (Выполнить) или нажмите `<F5>`, чтобы создать базу данных.

Создание таблиц

База данных `AutoLot` содержит пять таблиц: `Inventory`, `Makes`, `Customers`, `Orders` и `CreditRisks`.

Создание таблицы *Inventory*

После создания базы данных можно приступать к созданию таблиц. Первой таблицей будет *Inventory*. Откройте окно нового запроса и введите приведенные ниже команды SQL:

```
USE [AutoLot]
GO
CREATE TABLE [dbo].[Inventory](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [MakeId] [int] NOT NULL,
    [Color] [nvarchar](50) NOT NULL,
    [PetName] [nvarchar](50) NOT NULL,
    [TimeStamp] [timestamp] NULL,
    CONSTRAINT [PK_Inventory] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Щелкните на кнопке Run (или нажмите <F5>), чтобы создать таблицу *Inventory*.

Создание таблицы *Makes*

Таблица *Inventory* хранит внешний ключ в (пока еще не созданной) таблице *Makes*. Создайте новый запрос и введите следующие команды SQL для создания таблицы *Makes*:

```
USE [AutoLot]
GO
CREATE TABLE [dbo].[Makes](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](50) NOT NULL,
    [TimeStamp] [timestamp] NULL,
    CONSTRAINT [PK_Makes] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) ON [PRIMARY]
) ON [PRIMARY]
GO
```

Щелкните на кнопке Run (или нажмите <F5>), чтобы создать таблицу *Makes*.

Создание таблицы *Customers*

Таблица *Customers* будет хранить список покупателей. Создайте новый запрос и введите представленные далее команды SQL:

```
USE [AutoLot]
GO
CREATE TABLE [dbo].[Customers](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](50) NOT NULL,
    [LastName] [nvarchar](50) NOT NULL,
    [TimeStamp] [timestamp] NULL,
```

```

CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
(
    [Id] ASC
) ON [PRIMARY]
) ON [PRIMARY]
GO

```

Щелкните на кнопке Run (или нажмите <F5>), чтобы создать таблицу Customers.

Создание таблицы Orders

Создаваемая следующей таблица Orders будет использоваться для представления автомобилей, заказанных покупателями. Создайте новый запрос, введите показанные ниже команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```

USE [AutoLot]
GO
CREATE TABLE [dbo].[Orders] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [CustomerId] [int] NOT NULL,
    [CarId] [int] NOT NULL,
    [TimeStamp] [timestamp] NULL,
CONSTRAINT [PK_Orders] PRIMARY KEY CLUSTERED
(
    [Id] ASC
) ON [PRIMARY]
) ON [PRIMARY]
GO

```

Создание таблицы CreditRisks

Финальная таблица CreditRisks будет применяться для представления покупателей, связанных с кредитным риском. Создайте новый запрос, введите следующие команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```

USE [AutoLot]
GO
CREATE TABLE [dbo].[CreditRisks] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](50) NOT NULL,
    [LastName] [nvarchar](50) NOT NULL,
    [CustomerId] [int] NOT NULL,
    [TimeStamp] [timestamp] NULL,
CONSTRAINT [PK_CreditRisks] PRIMARY KEY CLUSTERED
(
    [Id] ASC
) ON [PRIMARY]
) ON [PRIMARY]
GO

```

Создание отношений между таблицами

В последующих разделах будут добавляться отношения внешнего ключа между взаимосвязанными таблицами.

Создание отношения между таблицами *Inventory* и *Makes*

Откройте окно нового запроса, введите показанные далее команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```
USE [AutoLot]
GO
CREATE NONCLUSTERED INDEX [IX_Inventory_MakeId] ON [dbo].[Inventory]
(
    [MakeId] ASC
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Inventory]
    WITH CHECK ADD CONSTRAINT [FK_Make_Inventory] FOREIGN
KEY([MakeId])
REFERENCES [dbo].[Makes] ([Id])
GO
ALTER TABLE [dbo].[Inventory] CHECK CONSTRAINT [FK_Make_Inventory]
GO
```

Создание отношения между таблицами *Inventory* и *Orders*

Откройте окно нового запроса, введите следующие команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```
USE [AutoLot]
GO
CREATE NONCLUSTERED INDEX [IX_Orders_CarId] ON [dbo].[Orders]
(
    [CarId] ASC
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Orders]
    WITH CHECK ADD CONSTRAINT [FK_Orders_Inventory] FOREIGN
KEY([CarId])
REFERENCES [dbo].[Inventory] ([Id])
GO
ALTER TABLE [dbo].[Orders] CHECK CONSTRAINT [FK_Orders_Inventory]
GO
```

Создание отношения между таблицами *Orders* и *Customers*

Откройте окно нового запроса, введите приведенные ниже команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```
USE [AutoLot]
GO
CREATE UNIQUE NONCLUSTERED INDEX [IX_Orders_CustomerId_CarId]
    ON [dbo].[Orders]
(
    [CustomerId] ASC,
    [CarId] ASC
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Orders]
    WITH CHECK ADD CONSTRAINT [FK_Orders_Customers] FOREIGN
```

```

KEY([CustomerId])
REFERENCES [dbo].[Customers] ([Id])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[Orders] CHECK CONSTRAINT [FK_Orders_Customers]
GO

```

Создание отношения между таблицами *Customers* и *CreditRisks*

Откройте окно нового запроса, введите следующие команды SQL и щелкните на кнопке Run (или нажмите <F5>):

```

USE [AutoLot]
GO
CREATE NONCLUSTERED INDEX [IX_CreditRisks_CustomerId]
    ON [dbo].[CreditRisks]
(
    [CustomerId] ASC
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[CreditRisks]
    WITH CHECK ADD CONSTRAINT [FK_CreditRisks_Customers]
FOREIGN KEY([CustomerId])
REFERENCES [dbo].[Customers] ([Id])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[CreditRisks] CHECK CONSTRAINT [FK_CreditRisks_Customers]
GO

```

На заметку! Наличие столбцов FirstName/LastName и отношение с таблицей преследует здесь только демонстрационные цели. В главе 23 они будут задействованы в более интересном сценарии.

Создание хранимой процедуры *GetPetName*

Позже в главе вы узнаете, как использовать ADO.NET для вызова хранимых процедур. Возможно, вам уже известно, что хранимые процедуры — это подпрограммы кода, хранящиеся внутри базы данных, которые выполняют какие-то действия. Подобно методам C# хранимые процедуры могут возвращать данные или просто работать с данными, ничего не возвращая. Добавьте одиночную хранимую процедуру, которая будет возвращать дружественное имя автомобиля на основе предоставленного carId. Откройте окно нового запроса и введите следующую команду SQL:

```

USE [AutoLot]
GO
CREATE PROCEDURE [dbo].[GetPetName]
@carID int,
@petName nvarchar(50) output
AS
SELECT @petName = PetName from dbo.Inventory where Id = @carID
GO

```

Щелкните на кнопке Run (или нажмите <F5>), чтобы создать хранимую процедуру.

Добавление тестовых записей

В отсутствие данных базы данных не особо интересны, поэтому удобно иметь сценарии, которые способны быстро загрузить тестовые записи в базу данных.

Записи таблицы Makes

Создайте новый запрос и выполните показанные далее операторы SQL для добавления записей в таблицу Makes:

```
USE [AutoLot]
GO
SET IDENTITY_INSERT [dbo].[Makes] ON
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (1, N'VW')
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (2, N'Ford')
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (3, N'Saab')
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (4, N'Yugo')
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (5, N'BMW')
INSERT INTO [dbo].[Makes] ([Id], [Name]) VALUES (6, N'Pinto')
SET IDENTITY_INSERT [dbo].[Makes] OFF
```

Записи таблицы Inventory

Чтобы добавить записи в таблицу Inventory, создайте новый запрос и выполните следующие операторы SQL:

```
USE [AutoLot]
GO
SET IDENTITY_INSERT [dbo].[Inventory] ON
GO
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (1, 1, N'Black', N'Zippy')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (2, 2, N'Rust', N'Rusty')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (3, 3, N'Black', N'Mel')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (4, 4, N'Yellow', N'Clunker')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (5, 5, N'Black', N'Bimmer')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (6, 5, N'Green', N'Hank')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (7, 5, N'Pink', N'Pinky')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (8, 6, N'Black', N'Pete')
INSERT INTO [dbo].[Inventory] ([Id], [MakeId], [Color], [PetName])
VALUES (9, 4, N'Brown', N'Brownie')
SET IDENTITY_INSERT [dbo].[Inventory] OFF
GO
```

Добавление тестовых записей в таблицу Customers

Чтобы добавить записи в таблицу Customers, создайте новый запрос и выполните представленные ниже операторы SQL:

```
USE [AutoLot]
GO
SET IDENTITY_INSERT [dbo].[Customers] ON
INSERT INTO [dbo].[Customers] ([Id], [FirstName], [LastName])
VALUES (1, N'Dave', N'Brenner')
INSERT INTO [dbo].[Customers] ([Id], [FirstName], [LastName])
VALUES (2, N'Matt', N'Walton')
INSERT INTO [dbo].[Customers] ([Id], [FirstName], [LastName])
VALUES (3, N'Steve', N'Hagen')
INSERT INTO [dbo].[Customers] ([Id], [FirstName], [LastName])
VALUES (4, N'Pat', N'Walton')
INSERT INTO [dbo].[Customers] ([Id], [FirstName], [LastName])
VALUES (5, N'Bad', N'Customer')
SET IDENTITY_INSERT [dbo].[Customers] OFF
```

Добавление тестовых записей в таблицу Orders

Теперь добавьте данные в таблицу Orders. Откройте окно нового запроса, введите следующую команду SQL и щелкните на кнопке Run (или нажмите <F5>):

```
USE [AutoLot]
GO
SET IDENTITY_INSERT [dbo].[Orders] ON
INSERT INTO [dbo].[Orders] ([Id], [CustomerId], [CarId]) VALUES (1, 1, 5)
INSERT INTO [dbo].[Orders] ([Id], [CustomerId], [CarId]) VALUES (2, 2, 1)
INSERT INTO [dbo].[Orders] ([Id], [CustomerId], [CarId]) VALUES (3, 3, 4)
INSERT INTO [dbo].[Orders] ([Id], [CustomerId], [CarId]) VALUES (4, 4, 7)
SET IDENTITY_INSERT [dbo].[Orders] OFF
```

Добавление тестовых записей в таблицу CreditRisks

Финальный шаг связан с добавлением данных в таблицу CreditRisks. Откройте окно нового запроса, введите следующую команду SQL и щелкните на кнопке Run (или нажмите <F5>):

```
USE [AutoLot]
GO
SET IDENTITY_INSERT [dbo].[CreditRisks] ON
INSERT INTO [dbo].[CreditRisks] ([Id], [FirstName], [LastName],
[CustomerId]) VALUES (1,
N'Bad', N'Customer', 5)
SET IDENTITY_INSERT [dbo].[CreditRisks] OFF
```

На этом создание базы данных AutoLot завершается. Конечно, она очень далека от базы данных реального приложения, но будет успешно удовлетворять всем нуждам текущей главы, а также добавляться в главах, посвященных Entity Framework Core. Располагая тестовой базой данных, можно приступить к погружению в детали, касающиеся модели фабрики поставщиков данных ADO.NET.

Модель фабрики поставщиков данных ADO.NET

Модель фабрики поставщиков данных .NET Core позволяет строить единую кодовую базу, используя обобщенные типы доступа к данным. Чтобы разобраться в реализации фабрики поставщиков данных, вспомните из табл. 21.1, что все классы внутри поставщика данных являются производными от тех же самых базовых классов, определенных внутри пространства имен System.Data.Common:

- `DbCommand` — абстрактный базовый класс для всех классов команд;
- `DbConnection` — абстрактный базовый класс для всех классов подключений;
- `DbDataAdapter` — абстрактный базовый класс для всех классов адаптеров данных;
- `DbDataReader` — абстрактный базовый класс для всех классов чтения данных;
- `DbParameter` — абстрактный базовый класс для всех классов параметров;
- `DbTransaction` — абстрактный базовый класс для всех классов транзакций.

Каждый поставщик данных, совместимый с .NET Core, содержит класс, производный от `System.Data.Common.DbProviderFactory`. В этом базовом классе определено несколько методов, которые извлекают объекты данных, специфичные для поставщика. Вот члены класса `DbProviderFactory`:

```
public abstract class DbProviderFactory
{
    public virtual bool CanCreateDataAdapter { get; };
    public virtual bool CanCreateCommandBuilder { get; };
    public virtual DbCommand CreateCommand();
    public virtual DbCommandBuilder CreateCommandBuilder();
    public virtual DbConnection CreateConnection();
    public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
    public virtual DbDataAdapter CreateDataAdapter();
    public virtual DbParameter CreateParameter();
    public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
}
```

Чтобы получить производный от `DbProviderFactory` тип для вашего поставщика данных, каждый поставщик предоставляет статическое свойство, используемое для возвращения корректного типа. Для возвращения версии SQL Server поставщика `DbProviderFactory` применяйте следующий код:

```
// Получить фабрику для поставщика данных SQL.
DbProviderFactory sqlFactory =
    Microsoft.Data.SqlClient.SqlClientFactory.Instance;
```

Чтобы сделать программу более универсальной, вы можете создать фабрику `DbProviderFactory`, которая возвращает конкретную разновидность `DbProviderFactory` на основе настройки в файле `appsettings.json` для приложения. Вскоре вы узнаете, как это делать, а пока после получения фабрики для поставщика данных можно получить связанные с ним объекты данных (например, объекты подключений, команд и чтения данных).

Полный пример фабрики поставщиков данных

В качестве завершенного примера создайте новый проект консольного приложения C# (по имени `DataProviderFactory`), которое выводит инвентарный список автомобилей из базы данных `AutoLot`. В начальном примере логика доступа к данным будет жестко закодирована прямо в сборке `DataProviderFactory.exe` (чтобы излишне не усложнять код). По мере изучения материалов настоящей главы вы узнаете более эффективные способы решения задачи.

Начните с добавления нового элемента `ItemGroup` и пакетов Microsoft.Extensions.Configuration.Json, System.Data.Common, System.Data.Odbc, System.Data.OleDb и Microsoft.Data.SqlClient в файл проекта:

```
dotnet add DataProviderFactory package Microsoft.Data.SqlClient
dotnet add DataProviderFactory package System.Data.Common
dotnet add DataProviderFactory package System.Data.Odbc
dotnet add DataProviderFactory package System.Data.OleDb
dotnet add DataProviderFactory
    package Microsoft.Extensions.Configuration.Json
```

Определите символ условной компиляции PC (в случае применения Windows):

```
<PropertyGroup>
  <DefineConstants>PC</DefineConstants>
</PropertyGroup>
```

Далее добавьте новый файл по имени `DataProviderEnum.cs` и модифицируйте его код, как показано ниже:

```
namespace DataProviderFactory
{
    // OleDb поддерживается только в Windows, но не в .NET Core.
    enum DataProviderEnum
    {
       SqlServer,
        #if PC
        OleDb,
        #endif
        Odbc
    }
}
```

Добавьте в проект новый файл JSON по имени `appsettings.json` и измените его содержимое следующим образом (обновите строки подключения в соответствии с имеющейся средой):

```
{
    "ProviderName": "SqlServer",
    // "ProviderName": "OleDb",
    // "ProviderName": "Odbc",
    "SqlServer": {
        // Для localdb используйте @"Data Source=(localdb) \
        // mssqllocaldb;Integrated Security=true;
        // Initial Catalog=AutoLot"
        "ConnectionString": "Data Source=.,5433;User Id=sa;
        Password=P@ssw0rd;Initial Catalog=AutoLot"
    },
    "Odbc": {
        // Для localdb используйте @"Driver={ODBC Driver 17 for SQL Server};
        Server=(localdb)\mssqllocaldb;Database=AutoLot;Trusted_Connection=True";
        "ConnectionString": "Driver={ODBC Driver 17 for SQL Server};
        Server=localhost,5433;Database=AutoLot;UID=sa;Pwd=P@ssw0rd;"
    },
    "OleDb": {
```

88 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
// Для localdb используйте @"Provider=SQLNCLI11;
// Data Source=(localdb)\mssqllocaldb;Initial
// Catalog=AutoLot;Integrated Security=SSPI"),
// "ConnectionString": "Provider=SQLNCLI11;Data Source=.,5433;
User Id=sa;Password=P@ssw0rd;
Initial Catalog=AutoLot;"}
}
```

Сообщите MSBuild о необходимости копировать файл JSON в выходной каталог при каждой компиляции. Модифицируйте файл проекта, как показано ниже:

```
<ItemGroup>
<None Update="appsettings.json">
<CopyToOutputDirectory>Always</CopyToOutputDirectory>
</None>
</ItemGroup>
```

На заметку! Элемент CopyToOutputDirectory чувствителен к наличию пробельных символов. Убедитесь, что пробелы вокруг слова Always отсутствуют.

Теперь, располагая подходящим файлом appsettings.json, вы можете читать значения provider и connectionString с использованием конфигурации .NET Core. Начните с обновления операторов using в верхней части файла Program.cs:

```
using System;
using System.Data.Common;
using System.Data.Odbc;
#if PC
    using System.Data.OleDb;
#endif
using System.IO;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;
```

Очистите весь код в Program.cs и добавьте взамен следующий код:

```
using System;
using System.Data.Common;
using System.Data.Odbc;
#if PC
    using System.Data.OleDb;
#endif
using System.IO;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;
using DataProviderFactory;

Console.WriteLine("***** Fun with Data Provider Factories *****\n");
var (provider, connectionString) = GetProviderFromConfiguration();
DbProviderFactory factory = GetDbProviderFactory(provider);

// Теперь получить объект подключения.
using (DbConnection connection = factory.CreateConnection())
{
```

```

if (connection == null)
{
    Console.WriteLine($"Unable to create the connection object");
    // Не удалось создать объект подключения
    return;
}

Console.WriteLine($"Your connection object is a: {connection.
GetType().Name}");
connection.ConnectionString = connectionString;
connection.Open();

// Создать объект команды.
DbCommand command = factory.CreateCommand();
if (command == null)
{
    Console.WriteLine($"Unable to create the command object");
    // Не удалось создать объект команды
    return;
}

Console.WriteLine($"Your command object is a: {command.GetType().Name}");
command.Connection = connection;
command.CommandText =
    "Select i.Id, m.Name From Inventory i inner join Makes m on m.Id =
i.MakeId ";

// Вывести данные с помощью объекта чтения данных.
using (DbDataReader dataReader = command.ExecuteReader())
{
    Console.WriteLine($"Your data reader object is a: {dataReader.
GetType().Name}");
    Console.WriteLine("\n***** Current Inventory *****");
    while (dataReader.Read())
    {
        Console.WriteLine($"-> Car #{dataReader["Id"]} is a
{dataReader["Name"]}");
    }
}
}

Console.ReadLine();

```

Добавьте приведенный далее код в конец файла Program.cs. Эти методы читают конфигурацию, устанавливают корректное значение DataProviderEnum, получают строку подключения и возвращают экземпляр DbProviderFactory:

```

static DbProviderFactory GetDbProviderFactory(DataProviderEnum provider)
    => provider switch
{
    DataProviderEnum.SqlServer => SqlClientFactory.Instance,
    DataProviderEnum.Odbc => OdbcFactory.Instance,
#if PC
    DataProviderEnum.OleDb => OleDbFactory.Instance,
#endif
    _ => null
};

```

```

static (DataProviderEnum Provider, string ConnectionString)
    GetProviderFromConfiguration()
{
    IConfiguration config = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", true, true)
        .Build();
    var providerName = config["ProviderName"];
    if (Enum.TryParse<DataProviderEnum>
        (providerName, out DataProviderEnum provider))
    {
        return (provider, config[$"{providerName}:ConnectionString"]);
    };
    throw new Exception("Invalid data provider value supplied.");
}

```

Обратите внимание, что в целях диагностики с помощью служб рефлексии выводятся имена лежащих в основе объектов подключения, команды и чтения данных. В результате запуска приложения в окне консоли отобразятся текущие данные из таблицы Inventory базы данных AutoLot:

```

***** Fun with Data Provider Factories *****

Your connection object is a: SqlConnection
Your command object is a: SqlCommand
Your data reader object is a: SqlDataReader

***** Current Inventory *****
-> Car #1 is a VW.
-> Car #2 is a Ford.
-> Car #3 is a Saab.
-> Car #4 is a Yugo.
-> Car #9 is a Yugo.
-> Car #5 is a BMW.
-> Car #6 is a BMW.
-> Car #7 is a BMW.
-> Car #8 is a Pinto.

```

Измените файл настроек, чтобы указать другого поставщика. Код выберет связанную строку подключения и произведет тот же вывод, что и ранее, исключая специфичную для типа информацию.

Конечно, в зависимости от опыта работы с ADO.NET у вас может не быть полно-го понимания того, что в действительности *делают* объекты подключений, команд и чтения данных. Не вдаваясь в детали, пока просто запомните, что модель фабрики поставщиков данных ADO.NET позволяет строить единственную кодовую базу, которая способна потреблять разнообразные поставщики данных в декларативной манере.

Потенциальный недостаток модели фабрики поставщиков данных

Хотя модель фабрики поставщиков данных характеризуется высокой мощностью, вы должны обеспечить применение в кодовой базе только типов и методов, общих для всех поставщиков, посредством членов абстрактных базовых классов. Следовательно, при разработке кодовой базы вы ограничены членами `DbConnection`, `DbCommand` и других типов из пространства имен `System.Data.Common`.

С учетом сказанного вы можете прийти к заключению, что такой обобщенный подход предотвращает прямой доступ к дополнительным возможностям отдельной СУБД. Если вы должны быть в состоянии обращаться к специфическим членам лежащего в основе поставщика (например, `SqlConnection`), то можете воспользоваться явным приведением:

```
if (connection is SqlConnection sqlConnection)
{
    // Вывести информацию об используемой версии SQL Server.
    WriteLine(sqlConnection.ServerVersion);
}
```

Однако в таком случае кодовая база становится чуть труднее в сопровождении (и менее гибкой), потому что придется добавить некоторое количество проверок времени выполнения. Тем не менее, если необходимо строить библиотеки доступа к данным наиболее гибким способом из числа возможных, тогда модель фабрики поставщиков данных предлагает замечательный механизм для решения такой задачи.

На заметку! Инфраструктура Entity Framework Core и ее поддержка внедрения зависимостей значительно упрощает построение библиотек доступа к данным, которым необходим доступ к разрозненным источникам данных.

Первый пример завершен, и теперь можно углубляться в детали работы с ADO.NET.

Погружение в детали объектов подключений, команд и чтения данных

Как было показано в предыдущем примере, ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, команд и чтения данных имеющегося поставщика данных. Для более глубокого понимания упомянутых объектов в ADO.NET будет создан расширенный пример.

В предыдущем примере демонстрировалось, что для подключения к базе данных и чтения записей посредством объекта чтения данных, необходимо было выполнить следующие шаги.

1. Создать, сконфигурировать и открыть объект подключения.
2. Создать и сконфигурировать объект команды, указав объект подключения в аргументе конструктора или через свойство `Connection`.
3. Вызвать метод `ExecuteReader()` на сконфигурированном объекте команды.
4. Обработать каждую запись с применением метода `Read()` объекта чтения данных.

Для начала создайте новый проект консольного приложения по имени `AutoLot`. `DataReader` и добавьте пакет `Microsoft.Data.SqlClient`. Ниже приведен полный код внутри `Program.cs` (с последующим анализом):

```
using System;
using Microsoft.Data.SqlClient;
Console.WriteLine("***** Fun with Data Readers *****\n");
```

```

// Создать и открыть подключение.
using (SqlConnection connection = new SqlConnection())
{
    connection.ConnectionString =
        @"Data Source=.,5433;User Id=sa;Password=P@ssw0rd;
Initial Catalog=AutoLot";
    connection.Open();
    // Создать объект команды SQL.
    string sql =
        @"Select i.id, m.Name as Make, i.Color, i.Petname
        FROM Inventory i
        INNER JOIN Makes m on m.Id = i.MakeId";
    SqlCommand myCommand = new SqlCommand(sql, connection);

    // Получить объект чтения данных с помощью ExecuteReader().
    using (SqlDataReader myDataReader = myCommand.ExecuteReader())
    {
        // Пройти в цикле по результатам.
        while (myDataReader.Read())
        {
            Console.WriteLine($"--> Make: {myDataReader["Make"]},
PetName: {myDataReader
            ["PetName"]}, Color: {myDataReader["Color"]}.");
        }
    }
    Console.ReadLine();
}

```

Работа с объектами подключений

При работе с поставщиком данных первым делом понадобится установить сеанс с источником данных, используя объект подключения (производного от `DbConnection` типа). Объекты подключений .NET Core обеспечиваются форматированной строкой подключения, которая содержит несколько пар "имя-значение", разделенных точками с запятой. Такая информация идентифицирует имя машины, к которой нужно подключиться, требуемые настройки безопасности, имя базы данных на машине и другие специфичные для поставщика сведения.

Из приведенного выше кода можно сделать вывод, что имя `Initial Catalog` относится к базе данных, с которой необходимо установить сеанс. Имя `Data Source` идентифицирует имя машины, где находится база данных. Здесь применяется строка `.,5433`, которая ссылается на хост-машину (точка соответствует `localhost`), и порт 5433, который представляет собой порт контейнера Docker, отображенный на порт SQL Server. Если бы вы использовали другой экземпляр, то определили бы свойство как `имя_машины,порт\экземпляр`. Например, `MY SERVER\SQL SERVER2019` означает, что `MY SERVER` — имя сервера, на котором функционирует SQL Server, что применяется стандартный порт и что `SQL SERVER2019` представляет собой имя экземпляра. Если машина является локальной по отношению к разработке, тогда можете использовать для имени сервера точку `(.)` или маркер `(localhost)`. В случае стандартного экземпляра SQL Server имя экземпляра не указывается. Скажем, если вы создаете базу данных `AutoLot` в установленной копии Microsoft SQL Server, настроенной как стандартный экземпляр на вашем локальном компьютере, то могли бы применять `"Data Source=localhost"`.

Кроме того, можно указать любое количество конструкций, которые представляют учетные данные безопасности. Если Integrated Security установлено в true, то для аутентификации и авторизации используется текущая учетная запись Windows.

Когда строка подключения готова, можно вызывать метод Open() для установления подключения к базе данных. В дополнение к членам ConnectionString, Open() и Close() объект подключения предоставляет несколько членов, которые позволяют конфигурировать дополнительные настройки подключения, такие как таймаут и транзакционная информация. В табл. 21.4 кратко описаны избранные члены базового класса DbConnection.

Таблица 21.4. Избранные члены типа DbConnection

Член	Описание
BeginTransaction()	Этот метод позволяет начать транзакцию базы данных
ChangeDatabase()	Этот метод изменяет базу данных, связанную с открытым подключением
ConnectionTimeout	Это свойство только для чтения возвращает промежуток времени, в течение которого происходит ожидание установления подключения, прежде чем будет сгенерирована ошибка (стандартное значение зависит от поставщика). Чтобы изменить стандартное значение, необходимо указать в строке подключения конструкцию Connect Timeout (например, Connect Timeout=30)
Database	Это свойство только для чтения возвращает имя базы данных, обслуживаемой объектом подключения
DataSource	Это свойство только для чтения возвращает местоположение базы данных, обслуживаемой объектом подключения
GetSchema()	Этот метод возвращает объект DataTable, содержащий информацию схемы из источника данных
State	Это свойство только для чтения возвращает текущее состояние подключения, представленное перечислением ConnectionState

Свойства типа DbConnection обычно по своей природе допускают только чтение и полезны, только если требуется получить характеристики подключения во время выполнения. Когда необходимо переопределить стандартные настройки, придется изменить саму строку подключения. Например, в следующей строке подключения время таймаута Connect Timeout устанавливается равным 30 секундам вместо стандартных 15 секунд (для SQL Server):

```
using(SqlConnection connection = new SqlConnection())
{
    connection.ConnectionString =
        @"Data Source=.,5433;User Id=sa;Password=P@ssw0rd;
Initial Catalog=AutoLot;Connect Timeout=30";
    connection.Open();
}
```

94 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

Следующий код выводит детали о переданной ему строке подключения SqlConnection:

```
static void ShowConnectionStatus(SqlConnection connection)
{
    // Вывести различные сведения о текущем объекте подключения.
    Console.WriteLine("***** Info about your connection *****");
    Console.WriteLine($"Database location:
{connection.DataSource}");           // Местоположение базы данных
    Console.WriteLine($"Database name: {connection.Database}");          // Имя базы данных
    Console.WriteLine($"Timeout:
{connection.ConnectionTimeout}");      // Таймаут
    Console.WriteLine($"Connection state:
{connection.State}\n");                // Состояние подключения
}
```

Большинство этих свойств понятно без объяснений, но свойство State требует специального упоминания. Ему можно присвоить любое значение из перечисления ConnectionState:

```
public enum ConnectionState
{
    Broken, Closed,
    Connecting, Executing,
    Fetching, Open
}
```

Однако допустимыми значениями ConnectionState будут только ConnectionState.Open, ConnectionState.Connecting и ConnectionState.Closed (остальные члены перечисления зарезервированы для будущего использования). Кроме того, закрывать подключение всегда безопасно, даже если его состоянием в текущий момент является ConnectionState.Closed.

Работа с объектами ConnectionStringBuilder

Работа со строками подключения в коде может быть утомительной, т.к. они часто представлены в виде строковых литералов, которые в лучшем случае трудно обрабатывать и контролировать на предмет ошибок. Совместимые с .NET Core поставщики данных поддерживают объекты построителей строк подключения, которые позволяют устанавливать пары "имя-значение" с применением строго типизированных свойств. Взгляните на следующую модификацию текущего кода:

```
var connectionStringBuilder = new SqlConnectionStringBuilder
{
    InitialCatalog = "AutoLot",
    DataSource = ".\.,5433",
    UserID = "sa",
    Password = "P@ssw0rd",
    ConnectTimeout = 30
};
connection.ConnectionString =
    connectionStringBuilder.ConnectionString;
```

В этой версии создается экземпляр класса `SqlConnectionStringBuilder`, соответствующим образом устанавливаются его свойства, после чего с использованием свойства `ConnectionString` получается внутренняя строка. Обратите внимание, что здесь применяется стандартный конструктор типа. При желании объект построителя строки подключения для поставщика данных можно также создать, передав в качестве отправной точки существующую строку подключения (что может быть удобно, когда значения динамически читаются из внешнего источника). После наполнения объекта начальными строковыми данными отдельные пары "имя-значение" можно изменять с помощью связанных свойств.

Работа с объектами команд

Теперь, когда вы лучше понимаете роль объекта подключения, следующей задачей будет выяснение, каким образом отправлять SQL-запросы базе данных. Тип `SqlCommand` (производный от `DbCommand`) является объектно-ориентированным представлением SQL-запроса, имени таблицы или хранимой процедуры. Тип команды указывается с использованием свойства `CommandType`, которое принимает любое значение из перечисления `CommandType`:

```
public enum CommandType
{
    StoredProcedure,
    TableDirect,
    Text // Стандартное значение.
}
```

При создании объекта команды SQL-запрос можно указывать как параметр конструктора или устанавливать свойство `CommandText` напрямую. Кроме того, когда создается объект команды, необходимо задать желаемое подключение. Его также можно указать в виде параметра конструктора либо с применением свойства `Connection`. Взгляните на следующий фрагмент кода:

```
// Создать объект команды посредством аргументов конструктора.
string sql =
    @"Select i.id, m.Name as Make, i.Color, i.Petname
      FROM Inventory i
      INNER JOIN Makes m on m.Id = i.MakeId";
SqlCommand myCommand = new SqlCommand(sql, connection);

// Создать еще один объект команды через свойства.
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = connection;
testCommand.CommandText = sql;
```

Учтите, что в текущий момент вы еще фактически не отправили SQL-запрос базе данных `AutoLot`, а только подготовили состояние объекта команды для будущего использования.

В табл. 21.5 описаны некоторые дополнительные члены типа `DbCommand`.

Таблица 21.5. Члены типа DbCommand

Член	Описание
CommandTimeout	Получает или устанавливает время ожидания, пока не завершится попытка выполнить команду и сгенерируется ошибка. Стандартное значение составляет 30 секунд
Connection	Получает или устанавливает объект DbConnection, применяемый текущим объектом DbCommand
Parameters	Получает коллекцию типов DbParameter, используемых для параметризованного запроса
Cancel()	Отменяет выполнение команды
ExecuteReader()	Выполняет запрос SQL и возвращает объект DbDataReader поставщика данных, который предоставляет допускающий только чтение доступ к результату запроса в прямом направлении
ExecuteNonQuery()	Выполняет оператор SQL, отличающийся от запроса (например, вставку, обновление, удаление или создание таблицы)
ExecuteScalar()	Легковесная версия метода ExecuteReader(), которая спроектирована специально для одноэлементных запросов (например, получение количества записей)
Prepare()	Создает подготовленную (или скомпилированную) версию команды для источника данных. Как вам может быть известно, подготовленный запрос выполняется несколько быстрее и удобен, когда один и тот же запрос необходимо выполнить много-кратно (обычно каждый раз с разными параметрами)

Работа с объектами чтения данных

После установления активного подключения и объекта команды SQL следующим действием будет отправка запроса источнику данных. Как вы наверняка догадались, это можно делать несколькими путями. Самый простой и быстрый способ получения информации из хранилища данных предлагает тип `DbDataReader` (реализующий интерфейс `IDataReader`). Вспомните, что объекты чтения данных представляют поток данных, допускающий только чтение в прямом направлении, который возвращает по одной записи за раз. Таким образом, объекты чтения данных полезны, только когда лежащему в основе хранилищу данных отправляются SQL-операторы выборки.

Объекты чтения данных удобны, если нужно быстро пройти по большому объему данных без необходимости иметь их представление в памяти. Например, в случае за-прашивания 20 000 записей из таблицы с целью их сохранения в текстовом файле помещение такой информации в объект `DataSet` приведет к значительному расходу памяти (поскольку `DataSet` хранит полный результат запроса в памяти).

Более эффективный подход предусматривает создание объекта чтения данных, который максимально быстро проходит по всем записям. Тем не менее, имейте в виду, что объекты чтения данных (в отличие от объектов адаптеров данных, которые рассматриваются позже) удерживают подключение к источнику данных открытым до тех пор, пока вы его явно не закроете.

Объекты чтения данных получаются из объекта команды с применением вызова `ExecuteReader()`. Объект чтения данных представляет текущую запись, прочитанную из базы данных. Он имеет метод индексатора (например, синтаксис `[]` в языке C#), который позволяет обращаться к столбцам текущей записи. Доступ к конкретному столбцу возможен либо по имени, либо по целочисленному индексу, начинающемуся с нуля.

В приведенном ниже примере использования объекта чтения данных задействован метод `Read()`, с помощью которого выясняется, когда достигнут конец записей (в случае чего он возвращает `false`). Для каждой прочитанной из базы данных записи с применением индексатора типа выводится модель, дружественное имя и цвет каждого автомобиля. Обратите внимание, что сразу после завершения обработки записей вызывается метод `Close()`, который освобождает объект подключения.

```
...
// Получить объект чтения данных посредством ExecuteReader().
using(SqlDataReader myDataReader = myCommand.ExecuteReader())
{
    // Пройти в цикле по результатам.
    while (myDataReader.Read())
    {
        WriteLine($"→ Make: { myDataReader["Make"] },
                  PetName: { myDataReader["PetName"] },
                  Color: { myDataReader["Color"] }.");
    }
}
ReadLine();
```

Индексатор объекта чтения данных перегружен для приема либо значения `string` (имя столбца), либо значения `int` (порядковый номер столбца). Таким образом, текущую логику объекта чтения можно сделать яснее (и избежать жестко закодированных строковых имен), внеся следующие изменения (обратите внимание на использование свойства `FieldCount`):

```
while (myDataReader.Read())
{
    for (int i = 0; i < myDataReader.FieldCount; i++)
    {
        Console.Write(i != myDataReader.FieldCount - 1
            ? $"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)}, "
            : $"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)} ");
    }
    Console.WriteLine();
}
```

Если в настоящий момент скомпилировать проект и запустить его на выполнение, то должен отобразиться список всех автомобилей из таблицы `Inventory` базы данных `AutoLot`. В следующем выводе показано несколько начальных записей:

```
***** Fun with Data Readers *****
***** Info about your connection *****
Database location: .,5433
Database name: AutoLot
Timeout: 30
Connection state: Open
```

```

id = 1, Make = VW, Color = Black, Petname = Zippy
id = 2, Make = Ford, Color = Rust, Petname = Rusty
id = 3, Make = Saab, Color = Black, Petname = Mel
id = 4, Make = Yugo, Color = Yellow, Petname = Clunker
id = 5, Make = BMW, Color = Black, Petname = Bimmer
id = 6, Make = BMW, Color = Green, Petname = Hank
id = 7, Make = BMW, Color = Pink, Petname = Pinky
id = 8, Make = Pinto, Color = Black, Petname = Pete
id = 9, Make = Yugo, Color = Brown, Petname = Brownie

```

Получение множества результирующих наборов с использованием объекта чтения данных

Объекты чтения данных могут получать несколько результирующих наборов с применением одиночного объекта команды. Например, если вы хотите получить все строки из таблицы *Inventory*, а также все строки из таблицы *Customers*, тогда можете указать два SQL-оператора *Select*, разделив их точкой с запятой:

```
sql += ";Select * from Customers;";
```

На заметку! Точка с запятой в начале строки опечаткой не является. В случае использования множества операторов они должны разделяться точками с запятой. И поскольку начальный оператор не содержал точку с запятой, она добавлена здесь в начало второго оператора.

После получения объекта чтения данных можно выполнить проход по каждому результирующему набору, используя метод *NextResult()*. Обратите внимание, что автоматически возвращается первый результирующий набор. Таким образом, если нужно прочитать все строки каждой таблицы, тогда можно построить следующую конструкцию итерации:

```

do
{
    while (myDataReader.Read())
    {
        for (int i = 0; i < myDataReader.FieldCount; i++)
        {
            Console.WriteLine($"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)}");
        }
        Console.WriteLine();
    }
    Console.WriteLine();
} while (myDataReader.NextResult());

```

К этому моменту вы уже должны лучше понимать функциональность, предлагаемую объектами чтения данных. Не забывайте, что объект чтения данных способен обрабатывать только SQL-операторы *Select*; его нельзя применять для изменения существующей таблицы базы данных с использованием запросов *Insert*, *Update* или *Delete*. Модификация существующей базы данных требует дальнейшего исследования объектов команд.

Работа с запросами создания, обновления и удаления

Метод `ExecuteReader()` извлекает объект чтения данных, который позволяет просматривать результаты SQL-оператора `Select` с помощью потока информации, допускающего только чтение в прямом направлении. Однако если необходимо отправить операторы SQL, которые в итоге модифицируют таблицу данных (или любой другой отличающийся от запроса оператор SQL, такой как создание таблицы либо выдача разрешений), то потребуется вызов метода `ExecuteNonQuery()` объекта команды. В зависимости от формата текста команды указанный единственный метод выполняет вставки, обновления и удаления.

На заметку! Говоря формально, “отличающийся от запроса” означает оператор SQL, который не возвращает результирующий набор. Таким образом, операторы `Select` являются запросами, тогда как `Insert`, `Update` и `Delete` — нет. С учетом сказанного метод `ExecuteNonQuery()` возвращает значение `int`, которое представляет количество строк, затронутых операторами, а не новый набор записей.

Все примеры взаимодействия с базами данных, рассмотренные в настоящей главе до сих пор, располагали только открытыми подключениями и применяли их для извлечения данных. Это лишь одна часть работы с базами данных; инфраструктура доступа к данным не приносила бы так много пользы, если бы полностью не поддерживала также функциональность создания, чтения, обновления и удаления (`create`, `read`, `update`, `delete` — CRUD). Далее вы научитесь пользоваться такой функциональностью, применяя вызовы `ExecuteNonQuery()`.

Начните с создания нового проекта библиотеки классов C# по имени `AutoLot`. DAL (сокращение от `AutoLot Data Access Layer` — уровень доступа к данным `AutoLot`), удалите стандартный файл класса и добавьте в проект пакет `Microsoft.Data.SqlClient`.

Перед построением класса, который будет управлять операциями с данными, сначала понадобится создать класс C#, представляющий запись из таблицы `Inventory` со связанный информацией `Make`.

Создание классов Car и CarViewModel

В современных библиотеках доступа к данным применяются классы (обычно называемые *моделями* или *сущностями*), которые используются для представления и транспортировки данных из базы данных. Кроме того, классы могут применяться для представления данных, которое объединяет две и большее количество таблиц, делая данные более значимыми. Сущностные классы используются при работе с каталогом базы данных (для операторов обновления), а классы модели представления применяются для отображения данных в осмысленной манере. В следующей главе вы увидите, что такие концепции являются основой инфраструктур объектно-реляционного отображения (object relational mapping — ORM) вроде Entity Framework Core, но пока вы просто собираетесь создать одну модель (для низкоуровневой строки хранилища) и одну модель представления (объединяющую строку хранилища и связанные данные в таблице `Makes`). Добавьте в проект новую папку по имени `Models` и поместите в нее два файла, `Car.cs` и `CarViewModel.cs`, со следующим кодом:

```
// Car.cs
namespace AutoLot.Dal.Models
{
    public class Car
    {
        public int Id { get; set; }
        public string Color { get; set; }
        public int MakeId { get; set; }
        public string PetName { get; set; }
        public byte[] TimeStamp { get; set; }
    }
}

// CarViewModel.cs
namespace AutoLot.Dal.Models
{
    public class CarViewModel : Car
    {
        public string Make { get; set; }
    }
}
```

На заметку! Если вы не знакомы с типом данных `TimeStamp` в SQL Server (который отображается на `byte[]` в C#), то беспокоиться об этом не стоит. Просто знайте, что он используется для проверки параллелизма на уровне строк и раскрывается вместе с Entity Framework Core.

Новые классы будут применяться вскоре.

Добавление класса `InventoryDal`

Далее добавьте новую папку по имени `DataOperations`. Поместите в нее файл класса по имени `InventoryDal.cs` и измените класс на `public`. В этом классе будут определены разнообразные члены, предназначенные для взаимодействия с таблицей `Inventory` базы данных `AutoLot`. Наконец, импортируйте следующие пространства имен:

```
using System;
using System.Collections.Generic;
using System.Data;
using AutoLot.Dal.Models;
using Microsoft.Data.SqlClient;
```

Добавление конструкторов

Создайте конструктор, который принимает строковый параметр (`connectionString`) и присваивает его значение переменной уровня класса. Затем создайте конструктор без параметров, передающий стандартную строку подключения другому конструктору. В итоге вызывающий код получит возможность изменения строки подключения, если стандартный вариант не подходит. Ниже показан соответствующий код:

```
namespace AuoLot.Dal.DataOperations
{
    public class InventoryDal
    {
        private readonly string _connectionString;
```

```

public InventoryDal() : this(
    @"Data Source=.,5433;User Id=sa;Password=P@ssw0rd;
Initial Catalog=AutoLot")
{
}
public InventoryDal(string connectionString)
    => _connectionString = connectionString;
}
}

```

Открытие и закрытие подключения

Добавьте переменную уровня класса, которая будет хранить подключение, применяемое кодом доступа к данным. Добавьте также два метода: один для открытия подключения (`OpenConnection()`) и еще один для закрытия подключения (`CloseConnection()`). В методе `CloseConnection()` проверьте состояние подключения и если оно не закрыто, тогда вызовите метод `Close()` на объекте подключения. Вот как выглядит код:

```

private SqlConnection _sqlConnection = null;
private void OpenConnection()
{
    _sqlConnection = new SqlConnection
    {
        ConnectionString = _connectionString
    };
    _sqlConnection.Open();
}

private void CloseConnection()
{
    if (_sqlConnection?.State != ConnectionState.Closed)
    {
        _sqlConnection?.Close();
    }
}

```

Ради краткости в большинстве методов класса `InventoryDal` не будут применяться блоки `try/catch` для обработки возможных исключений, равно как не будут генерироваться специальные исключения для сообщения о разнообразных проблемах при выполнении (скажем, неправильно сформированная строка подключения). Если бы строилась библиотека доступа к данным производственного уровня, то определенно пришлось бы использовать приемы структурированной обработки исключений (как объяснялось в главе 7), чтобы учесть любые аномалии времени выполнения.

Добавление реализации `IDisposable`

Добавьте к определению класса интерфейс `IDisposable`:

```

public class InventoryDal : IDisposable
{
    ...
}

```

Затем реализуйте шаблон освобождения, вызывая `Dispose()` на объекте `SqlConnection`:

```

bool _disposed = false;
protected virtual void Dispose(bool disposing)
{
    if (_disposed)
    {
        return;
    }
    if (disposing)
    {
        _sqlConnection.Dispose();
    }
    _disposed = true;
}
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

```

Добавление методов выборки

Для начала объедините имеющиеся сведения об объектах команд, чтения данных и обобщенных коллекциях, чтобы получить записи из таблицы Inventory. Как было показано в начале главы, объект чтения данных в поставщике делает возможной выборку записей с использованием механизма, который реализует только чтение в прямом направлении с помощью метода Read(). В этом примере свойство CommandBehavior класса DataReader настроено на автоматическое закрытие подключения, когда закрывается объект чтения данных. Метод GetAllInventory() возвращает экземпляр List<CarViewModel>, представляющий все данные в таблице Inventory:

```

public List<CarViewModel> GetAllInventory()
{
    OpenConnection();
    // Здесь будут храниться записи.
    List<CarViewModel> inventory = new List<CarViewModel>();
    // Подготовить объект команды.
    string sql =
        @"SELECT i.Id, i.Color, i.PetName, m.Name as Make
        FROM Inventory i
        INNER JOIN Makes m on m.Id = i.MakeId";
    using SqlCommand command =
        new SqlCommand(sql, _sqlConnection)
    {
        CommandType = CommandType.Text
    };
    command.CommandType = CommandType.Text;
    SqlDataReader dataReader =
        command.ExecuteReader(CommandBehavior.CloseConnection);
    while (dataReader.Read())
    {
        inventory.Add(new CarViewModel
        {
            Id = (int)dataReader["Id"],

```

```

        Color = (string)dataReader["Color"],
        Make = (string)dataReader["Make"],
        PetName = (string)dataReader["PetName"]
    );
}
dataReader.Close();
return inventory;
}

```

Следующий метод выборки получает одиночный объект CarViewModel на основе значения CarId:

```

public CarViewModel GetCar(int id)
{
    OpenConnection();
    CarViewModel car = null;
    // Параметры должны применяться по причинам, связанным с безопасностью.
    string sql =
        $"SELECT i.Id, i.Color, i.PetName,m.Name as Make
        FROM Inventory i
        INNER JOIN Makes m on m.Id = i.MakeId
        WHERE i.Id = {id}";
    using SqlCommand command =
        new SqlCommand(sql, _sqlConnection)
    {
        CommandType = CommandType.Text
    };
    SqlDataReader dataReader =
        command.ExecuteReader(CommandBehavior.CloseConnection);
    while (dataReader.Read())
    {
        car = new CarViewModel
        {
            Id = (int) dataReader["Id"],
            Color = (string) dataReader["Color"],
            Make = (string) dataReader["Make"],
            PetName = (string) dataReader["PetName"]
        };
    }
    dataReader.Close();
    return car;
}

```

На заметку! Помещение пользовательского ввода внутрь низкоуровневых операторов SQL, как делалось здесь, обычно считается неудачной практикой. Позже в главе код будет модифицирован для использования параметров.

Вставка новой записи об автомобиле

Вставка новой записи в таблицу `Inventory` сводится к построению SQL-оператора `Insert` (на основе пользовательского ввода), открытию подключения, вызову метода `ExecuteNonQuery()` с применением объекта команды и закрытию подключения. Увидеть вставку в действии можно, добавив к типу `InventoryDal` открытый метод по имени `InsertAuto()`, который принимает три параметра, отображаемые на не-

104 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

связанные с идентичностью столбцы таблицы `Inventory` (`Color`, `Make` и `PetName`). Указанные аргументы используются при форматировании строки для вставки новой записи. И, наконец, для выполнения итогового оператора SQL применяется объект `SqlConnection`.

```
public void InsertAuto(string color, int makeId, string petName)
{
    OpenConnection();
    // Сформатировать и выполнить оператор SQL.
    string sql = $"Insert Into Inventory (MakeId, Color, PetName)
        Values ('{makeId}', '{color}', '{petName}')";
    // Выполнить, используя наше подключение.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        command.ExecuteNonQuery();
    }
    CloseConnection();
}
```

Приведенный выше метод принимает три значения для `Car` и работает при условии, что вызывающий код передает значения в правильном порядке. Более совершенный метод использует `Car`, чтобы стать строго типизированным, гарантируя тем самым, что все свойства передаются методу в корректном порядке.

Создание строго типизированного метода `InsertCar()`

Добавьте в класс `InventoryDal` еще одну версию метода `InsertAuto()`, которая принимает в качестве параметра `Car`:

```
public void InsertAuto(Car car)
{
    OpenConnection();
    // Сформатировать и выполнить оператор SQL.
    string sql = "Insert Into Inventory (MakeId, Color, PetName) Values " +
        $"('{car.MakeId}', '{car.Color}', '{car.PetName}')";
    // Выполнить, используя наше подключение.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.CommandType = CommandType.Text;
        command.ExecuteNonQuery();
    }
    CloseConnection();
}
```

Добавление логики удаления

Удаление существующей записи не сложнее вставки новой записи. В отличие от метода `InsertAuto()` на этот раз вы узнаете о важном блоке `try/catch`, который обрабатывает возможную попытку удалить запись об автомобиле, уже заказанном кем-то из таблицы `Customers`. Стандартные параметры `INSERT` и `UPDATE` для внешних ключей по умолчанию предотвращают удаление зависимых записей в связанных таблицах. Когда предпринимается попытка подобного удаления, генерируется исключение `SqlException`.

В реальной программе была бы предусмотрена логика обработки такой ошибки, но в рассматриваемом примере просто генерируется новое исключение. Добавьте в класс InventoryDal следующий метод:

```
public void DeleteCar(int id)
{
    OpenConnection();
    // Получить идентификатор автомобиля, подлежащего удалению,
    // и удалить запись о нем.
    string sql = $"Delete from Inventory where Id = '{id}'";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        try
        {
            command.CommandType = CommandType.Text;
            command.ExecuteNonQuery();
        }
        catch (SqlException ex)
        {
            Exception error = new Exception("Sorry! That car is on order!", ex);
            // Этот автомобиль заказан!
            throw error;
        }
    }
    CloseConnection();
}
```

Добавление логики обновления

Когда речь идет об обновлении существующей записи в таблице `Inventory`, первым делом потребуется решить, какие характеристики будет позволено изменять вызывающему коду: цвет автомобиля, его дружественное имя, модель или все перечисленное? Один из способов предоставления вызывающему коду полной гибкости заключается в определении метода, принимающего параметр типа `string`, который представляет любой оператор SQL, но в лучшем случае это сопряжено с риском.

В идеале лучше иметь набор методов, которые позволяют вызывающему коду обновлять запись разнообразными способами. Тем не менее, определите для такой простой библиотеки доступа к данным единственный метод, который дает вызывающему коду возможность обновить дружественное имя указанного автомобиля:

```
public void UpdateCarPetName(int id, string newPetName)
{
    OpenConnection();
    //Получить идентификатор автомобиля для модификации дружественного имени
    string sql = $"Update Inventory Set PetName = '{newPetName}'"
    Where Id = '{id}'";
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        command.ExecuteNonQuery();
    }
    CloseConnection();
}
```

Работа с параметризованными объектами команд

В настоящий момент внутри логики вставки, обновления и удаления для типа `InventoryDal` используются жестко закодированные строковые литералы, представляющие каждый запрос SQL. В параметризованных запросах параметры SQL являются объектами, а не простыми порциями текста. Трактовка запросов SQL в более объектно-ориентированной манере помогает сократить количество опечаток (учитывая, что свойства строго типизированы). Вдобавок параметризованные запросы обычно выполняются значительно быстрее запросов в виде строковых литералов, т.к. они подвергаются разбору только однажды (а не каждый раз, когда строка с запросом SQL присваивается свойству `CommandText`). Параметризованные запросы также содействуют в защите против атак внедрением в SQL (хорошо известная проблема безопасности доступа к данным).

Для поддержки параметризованных запросов объекты команд ADO.NET содержат коллекцию индивидуальных объектов параметров. По умолчанию коллекция пуста, но в нее можно вставить любое количество объектов параметров, которые отображаются на параметры-заполнители в запросе SQL. Чтобы ассоциировать параметр внутри запроса SQL с членом коллекции параметров в объекте команды, параметр запроса SQL необходимо снабдить префиксом в виде символа @ (во всяком случае, когда применяется Microsoft SQL Server; не все СУБД поддерживают такую систему обозначений).

Указание параметров с использованием типа `DbParameter`

Перед построением параметризованного запроса вы должны ознакомиться с типом `DbParameter` (который является базовым классом для объекта параметра поставщика). Класс `DbParameter` поддерживает несколько свойств, которые позволяют конфигурировать имя, размер и тип параметра, а также другие характеристики, включая направление движения параметра. Некоторые основные свойства типа `DbParameter` описаны в табл. 21.6.

Таблица 21.6. Основные свойства типа `DbParameter`

Свойство	Описание
<code>DbType</code>	Получает или устанавливает собственный тип данных параметра, представленный как тип данных CLR
<code>Direction</code>	Получает или устанавливает направление движения параметра: только для ввода, только для вывода, для ввода и для вывода или для возвращаемого значения
<code>IsNullable</code>	Получает или устанавливает признак, может ли параметр принимать значения <code>null</code>
<code>ParameterName</code>	Получает или устанавливает имя <code>DbParameter</code>
<code>Size</code>	Получает или устанавливает максимальный размер данных в байтах для параметра (полезно только для текстовых данных)
<code>Value</code>	Получает или устанавливает значение параметра

Давайте теперь посмотрим, как заполнять коллекцию совместимых с `DbParameter` объектов, содержащуюся в объекте команды, для чего переделаем методы `InventoryDal` для использования параметров.

Обновление метода GetCar()

В исходной реализации метода GetCar() при построении строки SQL для извлечения данных об автомобиле применяется интерполяция строк C#. Чтобы обновить метод GetCar(), создайте экземпляр SqlParameter с соответствующими значениями:

```
SqlParameter param = new SqlParameter
{
    ParameterName = "@carId",
    Value = id,
    SqlDbType = SqlDbType.Int,
    Direction = ParameterDirection.Input
};
```

Значение ParameterName должно совпадать с именем, используемым в запросе SQL (который будет модифицирован следующим), тип обязан соответствовать типу столбца базы данных, а направление зависит от того, применяется параметр для отправки данных в запрос (ParameterDirection.Input) или он предназначен для возвращения данных из запроса (ParameterDirection.Output). Параметры также могут определяться как InputOutput или ReturnValue (возвращаемое значение, например, из хранимой процедуры).

Модифицируйте строку SQL для использования имени параметра ("@carId") вместо интерполированной строки C# "{id}":

```
string sql =
@"
SELECT i.Id, i.Color, i.PetName, m.Name as Make
FROM Inventory i
INNER JOIN Makes m on m.Id = i.MakeId
WHERE i.Id = @CarId";
```

Последнее обновление связано с добавлением нового объекта параметра в коллекцию Parameters объекта команды:

```
command.Parameters.Add(param);
```

Обновление метода DeleteCar()

Аналогично в исходной реализации метода DeleteCar() применяется интерполяция строк C#. Чтобы модифицировать этот метод, создайте экземпляр SqlParameter с надлежащими значениями:

```
SqlParameter param = new SqlParameter
{
    ParameterName = "@carId",
    Value = id,
    SqlDbType = SqlDbType.Int,
    Direction = ParameterDirection.Input
};
```

Обновите строку SQL для использования имени параметра ("@carId"):

```
string sql = "Delete from Inventory where Id = @carId";
```

В заключение добавьте новый объект параметра в коллекцию Parameters объекта команды:

```
command.Parameters.Add(param);
```

Обновление метода *UpdateCarPetName ()*

Метод *UpdateCarPetName ()* требует предоставления двух параметров: одного для *Id* автомобиля и еще одного для нового значения *PetName*. Первый параметр создается в точности как в предыдущих двух примерах (за исключением отличающегося имени переменной), а второй параметр обеспечивает отображение на тип *NVarChar* базы данных (тип поля *PetName* из таблицы *Inventory*). Обратите внимание на установку значения *Size*. Важно, чтобы этот размер совпадал с размером поля базы данных, что обеспечит отсутствие проблем при выполнении команды:

```
SqlParameter paramId = new SqlParameter
{
    ParameterName = "@carId",
    Value = id,
    SqlDbType = SqlDbType.Int,
    Direction = ParameterDirection.Input
};
SqlParameter paramName = new SqlParameter
{
    ParameterName = "@petName",
    Value = newPetName,
    SqlDbType = SqlDbType.NVarChar,
    Size = 50,
    Direction = ParameterDirection.Input
};
```

Модифицируйте строку SQL для применения параметров:

```
string sql = $"Update Inventory Set PetName = @petName Where Id = @carId";
```

Последнее обновление касается добавления новых параметров в коллекцию *Parameters* объекта команды:

```
command.Parameters.Add(paramId);
command.Parameters.Add(paramName);
```

Обновление метода *InsertAuto ()*

Добавьте следующую версию метода *InsertAuto ()*, чтобы задействовать объекты параметров:

```
public void InsertAuto(Car car)
{
    OpenConnection();
    // Обратите внимание на "заполнители" в запросе SQL.
    string sql = "Insert Into Inventory" +
        "(MakeId, Color, PetName) Values" +
        "(@MakeId, @Color, @PetName)";

    // Эта команда будет иметь внутренние параметры.
    using (SqlCommand command = new SqlCommand(sql, _sqlConnection))
    {
        // Заполнить коллекцию параметров.
        SqlParameter parameter = new SqlParameter
        {
            ParameterName = "@MakeId",
            Value = car.MakeId,
```

```

        SqlDbType = SqlDbType.Int,
        Direction = ParameterDirection.Input
    };
    command.Parameters.Add(parameter);
    parameter = new SqlParameter
    {
        ParameterName = "@Color",
        Value = car.Color,
        SqlDbType = SqlDbType.NVarChar,
        Size = 50,
        Direction = ParameterDirection.Input
    };
    command.Parameters.Add(parameter);
    parameter = new SqlParameter
    {
        ParameterName = "@PetName",
        Value = car.PetName,
        SqlDbType = SqlDbType.NVarChar,
        Size = 50,
        Direction = ParameterDirection.Input
    };
    command.Parameters.Add(parameter);
    command.ExecuteNonQuery();
    CloseConnection();
}
}

```

В то время как построение параметризованного запроса часто требует большего объема кода, в результате получается более удобный способ для программной настройки операторов SQL и достигается лучшая производительность. Параметризованные запросы также чрезвычайно удобны, когда нужно запускать хранимые процедуры.

Выполнение хранимой процедуры

Вспомните, что *хранимая процедура* представляет собой именованный блок кода SQL, сохраненный в базе данных. Хранимые процедуры можно конструировать так, чтобы они возвращали набор строк либо скалярных типов данных или выполняли еще какие-то осмысленные действия (например, вставку, обновление или удаление записей); в них также можно предусмотреть любое количество необязательных параметров. Конечным результатом будет единица работы, которая ведет себя подобно типичной функции, но только находится в хранилище данных, а не в двоичном бизнес-объекте. В текущий момент в базе данных AutoLot определена единственная хранимая процедура по имени GetPetName.

Рассмотрим следующий (пока что) финальный метод типа InventoryDal, в котором вызывается хранимая процедура GetPetName:

```

public string LookUpPetName(int carId)
{
    OpenConnection();
    string carPetName;

    // Установить имя хранимой процедуры.
    using (SqlCommand command = new SqlCommand("GetPetName", _sqlConnection))
    {

```

110 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
command.CommandType = CommandType.StoredProcedure;
// Входной параметр.
SqlParameter param = new SqlParameter
{
    ParameterName = "@carId",
    SqlDbType = SqlDbType.Int,
    Value = carId,
    Direction = ParameterDirection.Input
};
command.Parameters.Add(param);

// Выходной параметр.
param = new SqlParameter
{
    ParameterName = "@petName",
    SqlDbType = SqlDbType.NVarChar,
    Size = 50,
    Direction = ParameterDirection.Output
};
command.Parameters.Add(param);

// Выполнить хранимую процедуру.
command.ExecuteNonQuery();

// Возвратить выходной параметр.
carPetName = (string)command.Parameters["@petName"].Value;
CloseConnection();
}
return carPetName;
}
```

С вызовом хранимых процедур связан один важный аспект: объект команды может представлять оператор SQL (по умолчанию) либо имя хранимой процедуры. Когда объекту команды необходимо сообщить о том, что он будет вызывать хранимую процедуру, потребуется указать имя этой процедуры (в аргументе конструктора или в свойстве CommandText) и установить свойство CommandType в CommandType.StoredProcedure. (В противном случае возникнет исключение времени выполнения, т.к. по умолчанию объект команды ожидает оператор SQL.)

Далее обратите внимание, что свойство Direction параметра @petName установлено в ParameterDirection.Output. Как и ранее, все объекты параметров добавляются в коллекцию параметров объекта команды.

После того, как хранимая процедура, запущенная вызовом метода ExecuteNonQuery(), завершила работу, можно получить значение выходного параметра, просмотрев коллекцию параметров объекта команды и применив соответствующее приведение:

```
// Возвратить выходной параметр.
carPetName = (string)command.Parameters["@petName"].Value;
```

К настоящему моменту вы располагаете простейшей библиотекой доступа к данным, которую можно использовать при построении клиента для отображения и редактирования данных. Вопросы создания графических пользовательских интерфейсов пока не обсуждались, поэтому мы протестируем полученную библиотеку доступа к данным с помощью нового консольного приложения.

Создание консольного клиентского приложения

Добавьте к решению AutoLot.Dal новый проект консольного приложения (по имени AutoLot.Client) и ссылку на проект AutoLot.Dal. Ниже приведены соответствующие CLI-команды dotnet (предполагается, что ваше решение называется Chapter21_AllProjects.sln):

```
dotnet new console -lang c# -n AutoLot.Client -o .\AutoLot.Client -f net5.0
dotnet sln .\Chapter21_AllProjects.sln add .\AutoLot.Client
dotnet add AutoLot.Client package Microsoft.Data.SqlClient
dotnet add AutoLot.Client reference AutoLot.Dal
```

В случае использования Visual Studio щелкните правой кнопкой мыши на имени решения и выберите в контекстном меню пункт Add⇒New Project (Добавить⇒Новый проект). Установите новый проект в качестве стартового (щелкнув правой кнопкой мыши на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Set as Startup Project (Установить как стартовый проект)). Это обеспечит запуск нового проекта при инициализации отладки в Visual Studio. Если вы применяете Visual Studio Code, тогда перейдите в каталог AutoLot.Test и запустите проект (когда наступит время) с использованием dotnet run.

Очистите код, сгенерированный в Program.cs, и поместите в начало файла Program.cs следующие операторы using:

```
using System;
using System.Linq;
using AutoLot.Dal;
using AutoLot.Dal.Models;
using AutoLot.Dal.DataOperations;
using System.Collections.Generic;
```

Чтобы задействовать AutoLot.Dal, замените код метода Main() показанным далее кодом:

```
InventoryDal dal = new InventoryDal();
List<CarViewModel> list = dal.GetAllInventory();
Console.WriteLine(" ***** All Cars ***** ");
Console.WriteLine("Id\tMake\tColor\tPet Name");
foreach (var item in list)
{
    Console.WriteLine($"{item.Id}\t{item.Make}\t{item.Color}\t{item.PetName}");
}
Console.WriteLine();
CarViewModel car =
    dal.GetCar(list.OrderBy(x=>x.Color).Select(x => x.Id).First());
Console.WriteLine(" ***** First Car By Color ***** ");
Console.WriteLine("CarId\tMake\tColor\tPet Name");
Console.WriteLine($"{car.Id}\t{car.Make}\t{car.Color}\t{car.PetName}");

try
{
    // Это потерпит неудачу из-за наличия связанных данных в таблице Orders.
    dal.DeleteCar(5);
    Console.WriteLine("Car deleted."); // Запись об автомобиле удалена.
}
```

```

catch (Exception ex)
{
    Console.WriteLine($"An exception occurred: {ex.Message}");
    // Сгенерировано исключение
}
dal.InsertAuto(new Car {Color = "Blue", MakeId = 5, PetName = "TowMonster"});
list = dal.GetAllInventory();
var newCar = list.First(x => x.PetName == "TowMonster");
Console.WriteLine(" ***** New Car ***** ");
Console.WriteLine("CarId\tMake\tColor\tPet Name");
Console.WriteLine($"{newCar.Id}\t{newCar.Make}\t{newCar.Color}\t{newCar.PetName}");
dal.DeleteCar(newCar.Id);
var petName = dal.LookUpPetName(car.Id);
Console.WriteLine(" ***** New Car ***** ");
Console.WriteLine($"Car pet name: {petName}");
Console.Write("Press enter to continue... ");
Console.ReadLine();

```

Понятие транзакций базы данных

Давайте завершим исследование ADO.NET рассмотрением концепции транзакций базы данных. Выражаясь просто, *транзакция* — это набор операций базы данных, которые успешно выполняются или терпят неудачу как единая группа. Если одна из операций отказывает, тогда осуществляется откат всех остальных операций, как будто ничего не происходило. Несложно предположить, что транзакции по-настоящему важны для обеспечения безопасности, достоверности и согласованности табличных данных.

Транзакции также важны в ситуациях, когда операция базы данных включает в себя взаимодействие с множеством таблиц или хранимых процедур (либо с комбинацией атомарных элементов базы данных). Классическим примером транзакции может служить процесс перевода денежных средств с одного банковского счета на другой. Например, если вам понадобилось перевести \$500 с депозитного счета на текущий чековый счет, то следующие шаги должны быть выполнены в транзакционной манере.

1. Банк должен снять \$500 с вашего депозитного счета.
2. Банк должен добавить \$500 на ваш текущий чековый счет.

Вряд ли бы вам понравилось, если бы деньги были сняты с депозитного счета, но не переведены (из-за какой-то ошибки со стороны банка) на текущий чековый счет, потому что вы попросту лишились бы \$500. Однако если поместить указанные шаги внутрь транзакции базы данных, тогда СУБД гарантирует, что все взаимосвязанные шаги будут выполнены как единое целое. Если любая часть транзакции откажет, то будет произведен откат всей операции в исходное состояние. С другой стороны, если все шаги выполняются успешно, то транзакция будет зафиксирована.

На заметку! Из литературы, посвященной транзакциям, вам может быть известно сокращение ACID. Оно обозначает четыре ключевые характеристики транзакций: атомарность (atomic; все или ничего), согласованность (consistent; данные остаются устойчивыми на протяжении транзакции), изоляция (isolated; транзакции не влияют друг на друга) и постоянство (durable; транзакции сохраняются и протоколируются в журнале).

В свою очередь платформа .NET Core поддерживает транзакции различными способами. Здесь мы рассмотрим объект транзакции поставщика данных ADO.NET (`SqlTransaction` в случае `Microsoft.Data.SqlClient`).

В дополнение к готовой поддержке транзакций внутри библиотек базовых классов .NET Core можно также использовать язык SQL имеющейся СУБД. Например, вы могли бы написать хранимую процедуру, в которой применяются операторы BEGIN TRANSACTION, ROLLBACK и COMMIT.

Основные члены объекта транзакции ADO.NET

Все транзакции, которые будут использоваться, реализуют интерфейс `IDbTransaction`. Как упоминалось в начале главы, интерфейс `IDbTransaction` определяет несколько членов:

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }
    void Commit();
    void Rollback();
}
```

Обратите внимание на свойство `Connection`, возвращающее ссылку на объект подключения, который инициировал текущую транзакцию (как вы вскоре увидите, объект транзакции получается из заданного объекта подключения). Метод `Commit()` вызывается, если все операции в базе данных завершились успешно, что приводит к сохранению в хранилище данных всех ожидающих изменений. И наоборот, метод `Rollback()` можно вызвать в случае генерации исключения времени выполнения, что информирует СУБД о необходимости проигнорировать все ожидающие изменения и оставить первоначальные данные незатронутыми.

На заметку! Свойство `IsolationLevel` объекта транзакции позволяет указать, насколько активно транзакция должна защищаться от действий со стороны других параллельно выполняющихся транзакций. По умолчанию транзакции полностью изолируются вплоть до их фиксаций.

Помимо членов, определенных в интерфейсе `IDbTransaction`, тип `SqlTransaction` определяет дополнительный член под названием `Save()`, который предназначен для определения точек сохранения. Такая концепция позволяет откатить отказанную транзакцию до именованной точки вместо того, чтобы осуществлять откат всей транзакции. При вызове метода `Save()` с использованием объекта `SqlTransaction` можно задавать удобный строковый псевдоним, а при вызове `Rollback()` этот псевдоним можно указывать в качестве аргумента для выполнения частичного отката. Вызов `Rollback()` без аргументов приводит к отмене всех ожидающих изменений.

Добавление метода транзакции в `InventoryDal`

Давайте посмотрим, как работать с транзакциями ADO.NET программным образом. Начните с открытия созданного ранее проекта библиотеки кода `AutoLot.Dal` и добавьте в класс `InventoryDal` новый открытый метод по имени `ProcessCreditRisk()`, предназначенный для работы с кредитными рисками. Метод будет искать клиента, в случае нахождения поместит его в таблицу `CreditRisks` и добавит к фамилии метку "(Credit Risk)".

114 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
public void ProcessCreditRisk(bool throwEx, int customerId)
{
    OpenConnection();
    // Найти имя текущего клиента по идентификатору.
    string fName;
    string lName;
    var cmdSelect = new SqlCommand(
        "Select * from Customers where Id = @customerId",
        _sqlConnection);
    SqlParameter paramId = new SqlParameter
    {
        ParameterName = "@customerId",
        SqlDbType = SqlDbType.Int,
        Value = customerId,
        Direction = ParameterDirection.Input
    };
    cmdSelect.Parameters.Add(paramId);
    using (var dataReader = cmdSelect.ExecuteReader())
    {
        if (dataReader.HasRows)
        {
            dataReader.Read();
            fName = (string) dataReader["FirstName"];
            lName = (string) dataReader["LastName"];
        }
        else
        {
            CloseConnection();
            return;
        }
    }
    cmdSelect.Parameters.Clear();
    // Создать объекты команды, представляющие каждый шаг операции.
    var cmdUpdate = new SqlCommand(
        "Update Customers set LastName = LastName + ' (CreditRisk)' "
        "where Id = @customerId",
        _sqlConnection);
    cmdUpdate.Parameters.Add(paramId);
    var cmdInsert = new SqlCommand(
        "Insert Into CreditRisks (CustomerId, FirstName, LastName)"
        "Values( @CustomerId, @FirstName, @LastName)",
        _sqlConnection);
    SqlParameter parameterId2 = new SqlParameter
    {
        ParameterName = "@CustomerId",
        SqlDbType = SqlDbType.Int,
        Value = customerId,
        Direction = ParameterDirection.Input
    };
    SqlParameter parameterFirstName = new SqlParameter
    {
        ParameterName = "@FirstName",
        Value = fName,
        SqlDbType = SqlDbType.NVarChar,
```

```

        Size = 50,
        Direction = ParameterDirection.Input
    };
SqlParameter parameterLastName = new SqlParameter
{
    ParameterName = "@LastName",
    Value = lName,
    SqlDbType = SqlDbType.NVarChar,
    Size = 50,
    Direction = ParameterDirection.Input
};

cmdInsert.Parameters.Add(parameterId2);
cmdInsert.Parameters.Add(parameterFirstName);
cmdInsert.Parameters.Add(parameterLastName);
// Это будет получено из объекта подключения.
SqlTransaction tx = null;
try
{
    tx = _sqlConnection.BeginTransaction();
    // Включить команды в транзакцию.
    cmdInsert.Transaction = tx;
    cmdUpdate.Transaction = tx;
    // Выполнить команды.
    cmdInsert.ExecuteNonQuery();
    cmdUpdate.ExecuteNonQuery();
    // Эмулировать ошибку.
    if (throwEx)
    {
        // Возникла ошибка, связанная с базой данных! Отказ транзакции...
        throw new Exception("Sorry! Database error! Tx failed...");
    }
    // Зафиксировать транзакцию!
    tx.Commit();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    // Любая ошибка приведет к откату транзакции.
    // Использовать условную операцию для проверки на предмет null.
    tx?.Rollback();
}
finally
{
    CloseConnection();
}
}

```

Здесь используется входной параметр типа `bool`, который указывает, нужно ли генерировать произвольное исключение при попытке обработки проблемного клиента. Такой прием позволяет эмулировать непредвиденные обстоятельства, которые могут привести к неудачному завершению транзакции. Понятно, что это делается лишь в демонстрационных целях; настоящий метод транзакции не должен позволять вызывающему процессу нарушать работу логики по своему усмотрению!

Обратите внимание на применение двух объектов `SqlCommand` для представления каждого шага транзакции, которая будет запущена. После получения имени и фамилии клиента на основе входного параметра `customerID` с помощью метода `BeginTransaction()` объекта подключения можно получить допустимый объект `SqlTransaction`. Затем (что очень важно) потребуется привлечь к участию каждый объект команды, присвоив его свойству `Transaction` полученного объекта транзакции. Если этого не сделать, то логика вставки и обновления не будет находиться в транзакционном контексте.

После вызова метода `ExecuteNonQuery()` на каждой команде генерируется исключение, если (и только если) значение параметра `bool` равно `true`. В таком случае происходит откат всех ожидающих операций базы данных. Если исключение не было генерировано, тогда в результате вызова `Commit()` оба шага будут зафиксированы в таблицах базы данных.

Тестирование транзакции базы данных

Выберите одного из клиентов, добавленных в таблицу `Customers` (например, `Dave Benner`, `Id = 1`). Добавьте в `Program.cs` внутри проекта `AutoLot.Client` новый метод по имени `FlagCustomer()`:

```
void FlagCustomer()
{
    Console.WriteLine("***** Simple Transaction Example *****\n");
    // Простой способ позволить транзакции успешно завершиться или отказать
    bool throwEx = true;
    Console.Write("Do you want to throw an exception (Y or N): ");
    // Хотите ли вы генерировать исключение?

    var userAnswer = Console.ReadLine();
    if (string.IsNullOrEmpty(userAnswer) ||
        userAnswer.Equals("N", StringComparison.OrdinalIgnoreCase))
    {
        throwEx = false;
    }
    var dal = new InventoryDal();

    // Обработать клиента 1 - ввести идентификатор клиента,
    // подлежащего перемещению.
    dal.ProcessCreditRisk(throwEx, 1);
    Console.WriteLine("Check CreditRisk table for results");
    // Результаты ищите в таблице CreditRisk
    Console.ReadLine();
}
```

Если вы запустите программу и укажете на необходимость генерации исключения, то обнаружите, что фамилия клиента в таблице `Customers` не изменилась, т.к. произошел откат всей транзакции. Однако если исключение не генерировалось, тогда окажется, что фамилия клиента в таблице `Customers` изменилась и была добавлена в таблицу `CreditRisks`.

Выполнение массового копирования с помощью ADO.NET

В случае, когда необходимо загрузить много записей в базу данных, показанные до сих пор методы будут довольно неэффективными. В SQL Server имеется средство, называемое **массовым копированием**, которое предназначено специально для таких сценариев, и в ADO.NET для него предусмотрена оболочка в виде класса `SqlBulkCopy`. В настоящем разделе главы объясняется, как выполнять массовое копирование с помощью ADO.NET.

Исследование класса `SqlBulkCopy`

Класс `SqlBulkCopy` имеет один метод, `WriteToServer()` (и его асинхронную версию `WriteToServerAsync()`), который обрабатывает список записей и помещает данные в базу более эффективно, чем последовательность операторов `Insert`, выполненная с помощью объектов команд. Метод `WriteToServer()` перегружен, чтобы принимать объект `DataTable`, объект `DataReader` или массив объектов `DataRow`. Придерживаясь тематики главы, мы собираемся использовать версию `WriteToServer()`, которая принимает `DataReader`, так что необходимо создать специальный класс чтения данных.

Создание специального класса чтения данных

Желательно, чтобы специальный класс чтения данных был обобщенным и содержал список моделей, которые нужно импортировать. Создайте в проекте `AutoLot.DAL` новую папку по имени `BulkImport`, в ней — новый файл интерфейса `IMyDataReader.cs`, реализующего `IDataReader`, со следующим кодом:

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

namespace AutoLot.Dal.BulkImport
{
    public interface IMyDataReader<T> : IDataReader
    {
        List<T> Records { get; set; }
    }
}
```

Далее реализуйте специальный класс чтения данных. Как вы уже видели, классы чтения данных содержат много частей, отвечающих за перемещение данных. Хорошая новость в том, что для `SqlBulkCopy` придется реализовать лишь несколько из них. Создайте новый файл класса по имени `MyDataReader.cs` и добавьте в него перечисленные ниже операторы `using`:

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

Сделайте класс открытым и запечатанным и обеспечьте реализацию классом интерфейса `IMyDataReader`. Добавьте конструктор для принятия записей и установки свойства:

118 Часть VI. Работа с файлами, сериализация объектов и доступ к данным

```
public sealed class MyDataReader<T> : IMyDataReader<T>
{
    public List<T> Records { get; set; }
    public MyDataReader(List<T> records)
    {
        Records = records;
    }
}
```

Предложите Visual Studio или Visual Studio Code самостоятельно реализовать все методы (либо скопировать их), что даст вам отправную точку для специального класса чтения данных. В рассматриваемом сценарии потребуется реализовать лишь члены, кратко описанные в табл. 21.7.

Таблица 21.7. Основные члены интерфейса `IDataReader` для класса `SqlBulkCopy`

Член	Описание
Read()	Получает следующую запись; возвращает <code>true</code> , если существует еще одна запись, или <code>false</code> , если достигнут конец списка
FieldCount	Получает общее количество полей в источнике данных
GetValue()	Получает значение поля на основе его порядковой позиции
GetSchemaTable()	Получает информацию о схеме для целевой таблицы

Начните с метода `Read()`, который возвращает `false`, если класс для чтения находится в конце списка, или `true` (с инкрементированием счетчика уровня класса), если конец списка еще не достигнут. Добавьте переменную уровня класса, которая будет хранить текущий индекс `List<T>`, и обновите метод `Read()`, как показано ниже:

```
public class MyDataReader<T> : IMyDataReader<T>
{
    ...
    private int _currentIndex = -1;
    public bool Read()
    {
        if (_currentIndex + 1 >= Records.Count)
        {
            return false;
        }
        _currentIndex++;
        return true;
    }
}
```

Каждый метод `GetXXX()` и свойство `FieldCount` требуют знания специфической модели, подлежащей загрузке. Вот как выглядит метод `GetValue()`, использующий `CarViewModel`:

```
public object GetValue(int i)
{
    Car currentRecord = Records[_currentIndex] as Car;
    return i switch
```

```

    {
        0 => currentRecord.Id,
        1 => currentRecord.MakeId,
        2 => currentRecord.Color,
        3 => currentRecord.PetName,
        4 => currentRecord.TimeStamp,
        _ => string.Empty,
    };
}
}

```

База данных содержит только четыре таблицы, но это означает необходимость в наличии четырех вариаций класса чтения данных. А подумайте о реальной производственной базе данных, в которой таблиц гораздо больше! Решить проблему можно более эффективно с применением рефлексии (см. главу 17) и LINQ to Objects (см. главу 13).

Добавьте переменные `readonly` для хранения значений `PropertyInfo` модели и словарь, который будет использоваться для хранения местоположения поля и имени таблицы в SQL Server. Модифицируйте конструктор, чтобы он принимал свойства обобщенного типа и инициализировал объект `Dictionary`. Ниже показан добавленный код:

```

private readonly PropertyInfo[] _propertyInfos;
private readonly Dictionary<int, string> _nameDictionary;
public MyDataReader(List<T> records)
{
    Records = records;
    _propertyInfos = typeof(T).GetProperties();
    _nameDictionary = new Dictionary<int, string>();
}

```

Модифицируйте конструктор, чтобы он принимал строку подключения `SQLConnection`, а также строки для имен схемы и таблицы, куда будут вставлены записи, и добавьте для этих значений переменные уровня класса:

```

private readonly SqlConnection _connection;
private readonly string _schema;
private readonly string _tableName;
public MyDataReader(List<T> records, SqlConnection connection,
                    string schema, string tableName)
{
    Records = records;
    _propertyInfos = typeof(T).GetProperties();
    _nameDictionary = new Dictionary<int, string>();
    _connection = connection;
    _schema = schema;
    _tableName = tableName;
}

```

Далее реализуйте метод `GetSchemaTable()`, который извлекает информацию SQL Server, касающуюся целевой таблицы:

```

public DataTable GetSchemaTable()
{
    using var schemaCommand =
        new SqlCommand($"SELECT * FROM {_schema}.({_tableName})",
                      _connection);
}

```

```

using var reader = schemaCommand.ExecuteReader(CommandBehavior.SchemaOnly);
return reader.GetSchemaTable();
}

```

Модифицируйте конструктор, чтобы использовать SchemaTable для создания словаря, который содержит поля целевой таблицы в порядке их следования внутри базы данных:

```

public MyDataReader(List<T> records, SqlConnection connection,
    string schema, string tableName)
{
    ...
    DataTable schemaTable = GetSchemaTable();
    for (int x = 0; x < schemaTable?.Rows.Count; x++)
    {
        DataRow col = schemaTable.Rows[x];
        var columnName = col.Field<string>("ColumnName");
        _nameDictionary.Add(x, columnName);
    }
}

```

Теперь показанные далее методы могут быть реализованы обобщенным образом, используя полученную посредством рефлексии информацию:

```

public int FieldCount => _propertyInfos.Length;
public object GetValue(int i)
=> _propertyInfos
    .First(x => x.Name.Equals(_nameDictionary[i],
        StringComparison.OrdinalIgnoreCase))
    .GetValue(Records[_currentIndex]);

```

Для справки ниже приведены остальные методы, которые должны присутствовать (но не реализованы):

```

public string GetName(int i) => throw new NotImplementedException();
public int GetOrdinal(string name) => throw new NotImplementedException();
public string GetDataTypeName(int i) => throw new NotImplementedException();
public Type GetFieldType(int i) => throw new NotImplementedException();
public int GetValues(object[] values) => throw new NotImplementedException();
public bool GetBoolean(int i) => throw new NotImplementedException();
public byte GetByte(int i) => throw new NotImplementedException();
public long GetBytes(int i, long fieldOffset, byte[] buffer,
    int bufferoffset, int length)
=> throw new NotImplementedException();
public char GetChar(int i) => throw new NotImplementedException();
public long GetChars(int i, long fieldoffset, char[] buffer,
    int bufferoffset, int length)
=> throw new NotImplementedException();
public Guid GetGuid(int i) => throw new NotImplementedException();
public short GetInt16(int i) => throw new NotImplementedException();
public int GetInt32(int i) => throw new NotImplementedException();
public long GetInt64(int i) => throw new NotImplementedException();
public float GetFloat(int i) => throw new NotImplementedException();
public double GetDouble(int i) => throw new NotImplementedException();
public string GetString(int i) => throw new NotImplementedException();
public decimal GetDecimal(int i) => throw new NotImplementedException();

```

```

public DateTime GetDateTime(int i) => throw new NotImplementedException();
public IDataReader GetData(int i) => throw new NotImplementedException();
public bool IsDBNull(int i) => throw new NotImplementedException();
object IDataRecord.this[int i] => throw new NotImplementedException();
object IDataRecord.this[string name] => throw new NotImplementedException();
public void Close() => throw new NotImplementedException();
public DataTable GetSchemaTable() => throw new NotImplementedException();
public bool NextResult() => throw new NotImplementedException();
public int Depth { get; }
public bool IsClosed { get; }
public int RecordsAffected { get; }

```

Выполнение массового копирования

Добавьте в папку BulkImport новый файл открытого статического класса по имени ProcessBulkImport.cs. Поместите в начало файла следующие операторы using:

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using Microsoft.Data.SqlClient;

```

Добавьте код для поддержки открытия и закрытия подключений (похожий на код в классе InventoryDal):

```

private const string ConnectionString =
    @"Data Source=.,5433;User Id=sa;Password=P@ssw0rd;
Initial Catalog=AutoLot";
private static SqlConnection _sqlConnection = null;
private static void OpenConnection()
{
    _sqlConnection = new SqlConnection
    {
        ConnectionString = ConnectionString
    };
    _sqlConnection.Open();
}
private static void CloseConnection()
{
    if (_sqlConnection?.State != ConnectionState.Closed)
    {
        _sqlConnection?.Close();
    }
}

```

Для обработки записей классу SqlBulkCopy требуется имя (и схема, если она отличается от dbo). После создания нового экземпляра SqlBulkCopy (с передачей объекта подключения) установите свойство DestinationTableName. Затем создайте новый экземпляр специального класса чтения данных, который хранит список объектов, подлежащих массовому копированию, и вызовите метод WriteToServer(). Ниже приведен код метода ExecuteBulkImport():

```

public static void ExecuteBulkImport<T>(IEnumerable<T> records,
    string tableName)
{
    OpenConnection();
    using SqlConnection conn = _sqlConnection;
    SqlBulkCopy bc = new SqlBulkCopy(conn)
    {
        DestinationTableName = tableName
    };
    var dataReader = new MyDataReader<T>(records.ToList(), _sqlConnection,
        "dbo", tableName);
    try
    {
        bc.WriteToServer(dataReader);
    }
    catch (Exception ex)
    {
        // Здесь должно что-то делаться.
    }
    finally
    {
        CloseConnection();
    }
}

```

Тестирование массового копирования

Возвратите в проект AutoLot.Client и добавьте в Program.cs следующие операторы using:

```

using AutoLot.Dal.BulkImport;
using System.Collections.Generic;

```

Добавьте в файл Program.cs новый метод по имени DoBulkCopy(). Создайте список объектов Car и передайте его вместе с именем таблицы методу ExecuteBulkImport(). Оставшаяся часть кода отображает результаты массового копирования.

```

void DoBulkCopy()
{
    Console.WriteLine(" ***** Do Bulk Copy ***** ");
    var cars = new List<Car>
    {
        new Car() {Color = "Blue", MakeId = 1, PetName = "MyCar1"}, 
        new Car() {Color = "Red", MakeId = 2, PetName = "MyCar2"}, 
        new Car() {Color = "White", MakeId = 3, PetName = "MyCar3"}, 
        new Car() {Color = "Yellow", MakeId = 4, PetName = "MyCar4"} 
    };
    ProcessBulkImport.ExecuteBulkImport(cars, "Inventory");
    InventoryDal dal = new InventoryDal();
    List<CarViewModel> list = dal.GetAllInventory();
    Console.WriteLine(" ***** All Cars ***** ");
    Console.WriteLine("CarId\tMake\tColor\tPet Name");
}

```

```
foreach (var item in list)
{
    Console.WriteLine(
        $"{{item.Id}}\t{{item.Make}}\t{{item.Color}}\t{{item.PetName}}");
}
Console.WriteLine();
}
```

Хотя добавление четырех новых объектов `Car` не показывает достоинства работы, связанной с применением класса `SqlBulkCopy`, вообразите себе попытку загрузки тысяч записей. Мы проделывали подобное с таблицей `Customers`, и время загрузки составляло считанные секунды, тогда как проход в цикле по всем записям занимал часы! Как и все в .NET Core, класс `SqlBulkCopy` — просто еще один инструмент, который должен находиться в вашем инструментальном наборе и использоваться в ситуациях, когда в этом есть смысл.

Резюме

Инфраструктура ADO.NET представляет собой собственную технологию доступа к данным платформы .NET Core. В настоящей главе было начато исследование роли поставщиков данных, которые по существу являются конкретными реализациями нескольких абстрактных базовых классов (из пространства имен `System.Data.Common`) и интерфейсных типов (из пространства имен `System.Data`). Вы видели, что с применением модели фабрики поставщиков данных ADO.NET можно построить кодовую базу, не зависящую от поставщика.

Вы также узнали, что с помощью объектов подключений, объектов транзакций, объектов команд и объектов чтения данных можно выбирать, обновлять, вставлять и удалять записи. Кроме того, было показано, что объекты команд поддерживают внутреннюю коллекцию параметров, которые можно использовать для обеспечения безопасности к типам в запросах SQL; они также удобны при запуске хранимых процедур.

Наконец, вы научились защищать код манипулирования данными с помощью транзакций и ознакомились с применением класса `SqlBulkCopy` для загрузки крупных объемов данных в базы данных SQL Server, используя ADO.NET.

ЧАСТЬ VII

Entity Framework Core

ГЛАВА 22

Введение в Entity Framework Core

В предыдущей главе были исследованы основы ADO.NET. Инфраструктура ADO.NET позволяет программистам приложений .NET (относительно прямолинейно) работать с реляционными данными, начиная с выхода первоначальной версии платформы .NET. В пакете обновлений .NET 3.5 Service Pack 1 компания Microsoft предложила новый компонент API-интерфейса ADO.NET под названием *Entity Framework* (EF).

Общая цель EF — предоставить возможность взаимодействия с данными из реляционных баз данных с использованием объектной модели, которая отображается на прямую на бизнес-объекты (или объекты предметной области) в создаваемом приложении. Например, вместо того, чтобы трактовать пакет данных как коллекцию строк и столбцов, вы можете оперировать с коллекцией строго типизированных объектов, называемых *сущностями*. Такие сущности хранятся в специализированных классах коллекций, поддерживающих LINQ, что позволяет выполнять операции доступа к данным в коде C#. Классы коллекций обеспечивают средства запрашивания хранилища данных с применением той же грамматики LINQ, которая была раскрыта в главе 13.

Подобно .NET Core инфраструктура Entity Framework Core представляет собой полностью переписанную инфраструктуру Entity Framework 6. Она построена на основе .NET Core, давая возможность инфраструктуре EF Core функционировать на множестве платформ. Переписывание EF Core позволило добавить к EF Core новые средства и улучшения в плане производительности, которые не получилось бы разумно реализовать в EF 6.

Воссоздание целой инфраструктуры с нуля требует внимательного анализа того, какие средства будут поддерживаться новой инфраструктурой, а от каких придется отказаться. Одним из средств EF 6, которые отсутствуют в EF Core (и вряд ли когда-либо будут добавлены), является поддержка визуального конструктора сущностей (Entity Designer). В EF Core поддерживается парадигма разработки “сначала код”. Если вы уже имели дело с упомянутой парадигмой, тогда можете проигнорировать приведенное предостережение.

На заметку! Инфраструктуру EF Core можно использовать с существующими базами данных, а также с пустыми и/или новыми базами данных. Оба механизма называют парадигмой “сначала код”, что вероятно нельзя считать самым удачным наименованием. Шаблоны сущностных классов и классов, производных от DbContext, могут быть созданы из существующей базы данных, а базы данных могут создаваться и обновляться из сущностных классов. В главах, посвященных EF Core, вы изучите оба подхода.

С каждым новым выпуском в инфраструктуру EF Core добавлялись дополнительные средства, которые присутствовали в EF 6, плюс совершенно новые средства, не существующие в EF 6. С выходом выпуска 3.1 список важных функций, отсутствующих в EF Core (в сравнении с EF 6), был значительно уменьшен, а с выходом выпуска 5.0 разрыв сократился еще больше. Фактически инфраструктура EF Core располагает всем необходимым для большинства проектов.

В этой и следующей главах вы ознакомитесь с доступом к данным с применением Entity Framework Core. Вы узнаете о том, как создавать модель предметной области, сопоставлять сущностные классы и свойства с таблицами и столбцами базы данных, реализовывать отслеживание изменений, использовать интерфейс командной строки (command-line interface — CLI) инфраструктуры EF Core для создания шаблонных классов и миграций, а также освоите роль класса `DbContext`. Вдобавок вы узнаете о связывании сущностей с помощью навигационных свойств, транзакций и проверки параллелизма и многих других функциональных средств.

К тому моменту, когда вы завершите изучение этих двух глав, у вас будет финальная версия уровня доступа к данным для базы данных `AutoLot`. Прежде чем заняться непосредственно инфраструктурой EF Core, давайте обсудим инструменты объектно-реляционного отображения в целом.

На заметку! Двух глав далеко не достаточно, чтобы охватить все аспекты инфраструктуры Entity Framework Core, т.к. ей посвящены целые книги (по объему сравнимые с настоящей). Цель предлагаемых глав — предложить вам практические знания, которые позволят приступить к применению EF Core для разработки своей линейки бизнес-приложений.

Инструменты объектно-реляционного отображения

Инфраструктура ADO.NET снабжает вас структурой, которая позволяет выбирать, вставлять, обновлять и удалять данные с помощью объектов подключений, команд и чтения данных. Тем не менее, такие аспекты ADO.NET вынуждают обходиться с извлеченными данными в манере, тесно связанной с физической схемой базы данных. В качестве примера вспомните, что при получении записей из базы данных вы открываете объект подключения, создаете и выполняете объект команды и затем используете объект чтения данных для прохода по записям с применением имен столбцов, зависящих от базы данных.

При работе с ADO.NET вы всегда обязаны помнить о физической структуре серверной базы данных. Вы должны знать схему каждой таблицы данных, создавать потенциально сложные запросы SQL для взаимодействия с таблицей (таблицами) данных, отслеживать изменения в извлеченных (или добавленных) данных и т.д. В итоге вы можете быть вынуждены записывать довольно многословный код C#, поскольку сам язык C# не позволяет работать непосредственно со схемой базы данных.

Хуже того, обычный способ создания физической базы данных прямо сосредоточен на конструкциях базы данных, таких как внешние ключи, представления, хранимые процедуры и нормализация данных, а не на объектно-ориентированном программировании.

Еще одним вопросом у разработчиков приложений, требующим решения, является отслеживание изменений. Получение данных из базы — один из этапов процесса, но любые изменения, добавления и/или удаления должны отслеживаться разработчиком, чтобы их можно было сохранить в хранилище данных.

Доступность инфраструктур *объектно-реляционного отображения* (*object-relational mapping* — ORM) в .NET значительно улучшила ситуацию с доступом к данным, управляя вместо разработчика большинством задач создания, чтения, обновления и удаления (*create, read, update, delete* — CRUD). Разработчик создает отображение между объектами .NET и реляционной базой данных, а инфраструктура ORM управляет подключениями, генерацией запросов, отслеживанием изменений и хранением данных. В итоге разработчик получает возможность целиком сосредоточиться на бизнес-потребностях приложения.

На заметку! Важно помнить, что инфраструктуры ORM не являются инструментами, которые с легкостью решат все проблемы. С каждым решением связаны компромиссы. Инфраструктуры ORM сокращают объем работы разработчикам, создающим уровни доступа к данным, но могут также привносить проблемы с производительностью и масштабированием в случае ненадлежащего применения. Используйте инфраструктуры ORM для операций CRUD и действуйте мощь своей базы данных для операций, основанных на множествах.

Хотя разные инфраструктуры ORM имеют небольшие отличия в том, как они работают, или каким образом применяются, все они по существу представляют собой одни и те же фрагменты и части, преследующие ту же самую цель — облегчить выполнение операций доступ к данным. Сущности являются классами, которые отображаются на таблицы базы данных. Специализированный тип коллекции содержит одну или большее количество сущностей. Механизм отслеживания изменений следит за состоянием объектов и любыми связанными с ними изменениями, добавлениями и/или удалениями, а центральная конструкция управляет операциями как руководитель.

Роль Entity Framework Core

“За кулисами” EF Core использует инфраструктуру ADO.NET, которая уже была исследована в предыдущей главе. Подобно любому взаимодействию ADO.NET с хранилищем данных EF Core применяет для этого поставщик данных ADO.NET. Прежде чем поставщик данных ADO.NET можно будет использовать в EF Core, его потребуется обновить для полной интеграции с EF Core. Из-за такой добавленной функциональности доступных поставщиков данных EF Core может оказаться меньше, чем поставщиков данных ADO.NET.

Преимущество инфраструктуры EF Core, применяющей шаблон поставщиков баз данных ADO.NET, заключается в том, что она позволяет объединять в одном проекте парадигмы доступа к данным EF Core и ADO.NET, расширяя ваши возможности. Например, в случае использования EF Core с целью предоставления подключения, схемы и имени таблицы для операций массового копирования задействуются возможности сопоставления EF Core и функциональность программы массового копирования, встроенная в ADO.NET. Такой смешанный подход делает EF Core просто еще одним инструментом в вашем арсенале.

Когда вы оцените объем связующего кода для базового доступа к данным, поддерживаемый инфраструктурой EF Core в согласованной и эффективной манере, по всей видимости, она станет вашим основным механизмом при доступе к данным.

На заметку! Многие сторонние СУБД (скажем, Oracle и MySQL) предлагают поставщики данных, осведомленные об инфраструктуре EF Core. Если вы имеете дело не с SQL Server, тогда обратитесь за детальными сведениями к разработчику СУБД или ознакомьтесь с перечнем доступных поставщиков данных EF Core по ссылке <https://docs.microsoft.com/ru-ru/ef/core/providers/>.

Инфраструктура EF Core лучше всего вписывается в процесс разработки в случае применения подходов в стиле “формы поверх данных” (или “API-интерфейс поверх данных”). Оптимальными для EF Core являются операции над небольшим количеством сущностей, использующие шаблон единицы работы с целью обеспечения согласованности. Она не очень хорошо подходит для выполнения крупномасштабных операций над данными вроде тех, что встречаются приложениях хранилищ данных типа “извлечение, трансформация, загрузка” (extract-transform-load — ETL) или в больших системах построения отчетов.

Строительные блоки Entity Framework Core

К главным компонентам EF Core относятся `DbContext`, `ChangeTracker`, специализированный тип коллекции `DbSet`, поставщики баз данных и сущности приложения. Для проработки примеров в текущем разделе создайте новый проект консольного приложения по имени `AutoLot.Samples` и добавьте к нему пакеты `Microsoft.EntityFrameworkCore`, `Microsoft.EntityFrameworkCore.Design` и `Microsoft.EntityFrameworkCore.SqlServer`:

```
dotnet new sln -n Chapter22_AllProjects
dotnet new console -lang c# -n AutoLot.Samples -o .\AutoLot.Samples -f net5.0
dotnet sln .\Chapter22_AllProjects.sln add .\AutoLot.Samples
dotnet add AutoLot.Samples package Microsoft.EntityFrameworkCore
dotnet add AutoLot.Samples package Microsoft.EntityFrameworkCore.Design
dotnet add AutoLot.Samples package Microsoft.EntityFrameworkCore.SqlServer
```

Класс `DbContext`

Класс `DbContext` входит в состав главных компонентов EF Core и предоставляет доступ к базе данных через свойство `Database`. Объект `DbContext` управляет экземпляром `ChangeTracker`, поддерживает виртуальный метод `OnModelCreating()` для доступа к текущему API-интерфейсу (Fluent API), хранит все свойства `DbSet<T>` и предлагает метод `SaveChanges()`, позволяющий сохранять данные в хранилище. Он применяется не напрямую, а через специальный класс, унаследованный от `DbContext`. Именно в этом классе размещены все свойства типа `DbSet<T>`.

В табл. 22.1 описаны некоторые часто используемые члены класса `DbContext`.

Таблица 22.1. Распространенные члены `DbContext`

Члены	Описание
<code>Database</code>	Обеспечивает доступ к связанной с базой данных информацией и функциональности, включая выполнение операторов SQL
<code>Model</code>	Метаданные о форме сущностей, отношениях между ними и их сопоставлении с базой данных. Примечание: с этим свойством обычно не взаимодействуют напрямую
<code>ChangeTracker</code>	Предоставляет доступ к информации и операциям для экземпляров сущностей, отслеживаемых этим объектом <code>DbContext</code>
<code>DbSet<T></code>	Не подлинный член <code>DbContext</code> , а свойства, добавленные к специальному производному от <code>DbContext</code> классу. Свойства имеют тип <code>DbSet<T></code> и применяются для запрашивания и сохранения экземпляров сущностей приложения. Запросы LINQ к свойствам <code>DbSet<T></code> транслируются в запросы SQL

Члены	Описание
Entry ()	Обеспечивает доступ к информации по отслеживанию изменений и операциям для сущности, таким как явная загрузка связанных сущностей либо изменение EntityState. Также может вызываться на неотслеживаемой сущности, чтобы изменить ее состояние на отслеживаемое
Set< TEntity > ()	Создает экземпляр свойства DbSet< T >, который может использоваться для запрашивания и сохранения данных
SaveChanges () / SaveChangesAsync ()	Сохраняет все изменения сущностей в базе данных и возвращает количество затронутых записей. Выполняется внутри транзакции (явной или неявной)
Add () / AddRange () Update () / UpdateRange () Remove () / RemoveRange ()	Методы для добавления, обновления и удаления экземпляров сущностей. Изменения сохраняются только после успешного выполнения метода SaveChanges (). Доступны также асинхронные версии. Примечание: хотя эти методы доступны в производном от DbContext классе, обычно они вызываются напрямую на свойствах DbSet< T >
Find ()	Ищет сущность типа с заданными значениями первичного ключа. Доступны также асинхронные версии. Примечание: хотя эти методы доступны в производном от DbContext классе, обычно они вызываются напрямую на свойствах DbSet< T >
Attach () / AttachRange ()	Начинает отслеживать сущность (или список сущностей). Доступны также асинхронные версии. Примечание: хотя эти методы доступны в производном от DbContext классе, обычно они вызываются напрямую на свойствах DbSet< T >
SavingChanges	Событие, инициируемое в начале вызова SaveChanges () / SaveChangesAsync ()
SavedChanges	Событие, инициируемое в конце вызова SaveChanges () / SaveChangesAsync ()
SaveChangesFailed	Событие, инициируемое в случае отказа вызова SaveChanges () / SaveChangesAsync ()
OnModelCreating ()	Вызывается при инициализации модели, но до ее финализации. В этот метод помещаются методы из Fluent API для финализации формы модели
OnConfiguring ()	Постройтель, используемый для создания или модификации параметров DbContext. Выполняется каждый раз, когда создается экземпляр DbContext. Примечание: рекомендуется применять не его, а DbContextOptions для конфигурирования экземпляра DbContext во время выполнения и использовать экземпляр реализации IDesignTimeDbContextFactory на этапе проектирования

Создание класса, производного от `DbContext`

Первый шаг в EF Core заключается в создании специального класса, унаследованного от `DbContext`. Затем добавляется конструктор, который принимает строго типизированный экземпляр `DbContextOptions` (рассматривается далее) и передает его конструктору базового класса:

```
namespace AutoLot.Samples
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) : base(options)
        {
        }
    }
}
```

Именно производный от `DbContext` класс применяется для доступа к базе данных и работает с сущностями, средством отслеживания изменений и всеми компонентами EF Core.

Конфигурирование экземпляра `DbContext`

Экземпляр `DbContext` конфигурируется с использованием экземпляра класса `DbContextOptions`. Экземпляр `DbContextOptions` создается с применением `DbContextOptionsBuilder`, т.к. класс `DbContextOptions` не рассчитан на создание экземпляров непосредственно в коде. Через экземпляр `DbContextOptionsBuilder` выбирается поставщик базы данных (наряду с любыми настройками, касающимися поставщика) и устанавливаются общие параметры экземпляра `DbContext` инфраструктуры EF Core (наподобие ведения журнала). Затем свойство `Options` внедряется в базовый класс `DbContext` во время выполнения.

Такая возможность динамического конфигурирования позволяет изменять настройки во время выполнения, просто выбирая разные параметры (скажем, поставщик MySQL вместо SQL Server) и создавая новый экземпляр производного класса `DbContext`.

Фабрика `DbContext` этапа проектирования

Фабрика `DbContext` этапа проектирования представляет собой класс, который реализует интерфейс `IDesignTimeDbContextFactory<T>`, где `T` — класс, производный от `DbContext`. Интерфейс `IDesignTimeDbContextFactory<T>` имеет один метод `CreateDbContext()`, который должен быть реализован для создания экземпляра производного класса `DbContext`.

В показанном ниже классе `ApplicationDbContextFactory` с помощью метода `CreateDbContext()` создается строго типизированный экземпляр `DbContext` `OptionsBuilder` для класса `ApplicationDbContext`, устанавливается поставщик баз данных `SQL Server` (с использованием строки подключения к экземпляру `Docker` из главы 21), после чего создается и возвращается новый экземпляр `ApplicationDbContext`:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
```

```

namespace AutoLot.Samples
{
    public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
    {
        public ApplicationDbContext CreateDbContext(string[] args)
        {
            var optionsBuilder =
                new DbContextOptionsBuilder<ApplicationContext>();
            var connectionString = @"server=.;5433;Database=AutoLotSamples;
User Id=sa;Password=P@ssw0rd;";
            optionsBuilder.UseSqlServer(connectionString);
            Console.WriteLine(connectionString);
            return new ApplicationDbContext(optionsBuilder.Options);
        }
    }
}

```

Интерфейс командной строки задействует фабрику контекстов, чтобы создать экземпляр производного класса `DbContext`, предназначенный для выполнения действий вроде создания и применения миграций базы данных. Поскольку фабрика является конструкцией этапа проектирования и не используется во время выполнения, строка подключения к базе данных разработки обычно будет жестко закодированной.

В версии EF Core 5 появилась возможность передавать методу `CreateDbContext()` аргументы из командной строки, о чем пойдет речь позже в главе.

Метод *OnModelCreating()*

Базовый класс `DbContext` открывает доступ к методу `OnModelCreating()`, который применяется для придания формы сущностям, используя Fluent API. Детали подробно раскрываются далее в главе, а пока добавьте в класс `ApplicationContext` следующий код:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Обращения к Fluent API.
    OnModelCreatingPartial(modelBuilder);
}
partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

```

Сохранение изменений

Чтобы заставить `DbContext` и `ChangeTracker` сохранить любые изменения, внесенные в отслеживаемые сущности, вызовите метод `SaveChanges()` (или `SaveChangesAsync()`) на экземпляре класса, производного от `DbContext`:

```

static void SampleSaveChanges()
{
    // Фабрика не предназначена для такого использования,
    // но это демонстрационный код.
    var context = new ApplicationDbContextFactory().CreateDbContext(null);

    // Внести какие-нибудь изменения.
    context.SaveChanges();
}

```

В оставшемся материале главы (и книги) вы обнаружите много примеров сохранения изменений.

Поддержка транзакций и точек сохранения

Исполняющая среда EF Core помещает каждый вызов `SaveChanges()` / `SaveChangesAsync()` внутрь неявной транзакции, использующей уровень изоляции базы данных. Чтобы добиться большей степени контроля, можете включить экземпляр производного класса `DbContext` в явную транзакцию. Для выполнения явной транзакции создайте транзакцию с применением свойства `Database` класса, производного от `DbContext`. Управляйте своими операциями обычным образом и затем предпримите фиксацию или откат транзакции. Ниже приведен фрагмент кода, где все демонстрируется:

```
using var trans = context.Database.BeginTransaction();
try
{
    // Создать, изменить, удалить запись.
    context.SaveChanges();
    trans.Commit();
}
catch (Exception ex)
{
    trans.Rollback();
}
```

В версии EF Core 5 были введены точки сохранения для транзакций EF Core. Когда вызывается метод `SaveChanges()` / `SaveChangesAsync()`, а транзакция уже выполняется, исполняющая среда EF Core создает в этой транзакции точку сохранения. Если вызов терпит неудачу, то откат транзакции происходит в точку сохранения, а не в начало транзакции. Точками сохранения можно также управлять в коде, вызывая методы `CreateSavePoint()` и `RollbackToSavepoint()` для транзакции:

```
using var trans = context.Database.BeginTransaction();
try
{
    // Создать, изменить, удалить запись.
    trans.CreateSavepoint("check point 1");
    context.SaveChanges();
    trans.Commit();
}
catch (Exception ex)
{
    trans.RollbackToSavepoint("check point 1");
}
```

Транзакции и стратегии выполнения

В случае активной стратегии выполнения (как при использовании `EnableRetryOnFailure()`) перед созданием явной транзакции вы должны получить ссылку на текущую стратегию выполнения, которая применяется EF Core. Затем вызовите на этой стратегии метод `Execute()`, чтобы создать явную транзакцию:

```

var strategy = context.Database.CreateExecutionStrategy();
strategy.Execute(() =>
{
    using var trans = context.Database.BeginTransaction();
    try
    {
        actionToExecute();
        trans.Commit();
    }
    catch (Exception ex)
    {
        trans.Rollback();
    }
});

```

События SavingChanges/SavedChanges

В версии EF Core 5 появились три новых события, которые инициируются методами `SaveChanges()` / `SaveChangesAsync()`. Событие `SavingChanges` запускается при вызове `SaveChanges()`, но перед выполнением операторов SQL в хранилище данных, а событие `SavedChanges` — после завершения работы метода `SaveChanges()`. В следующем (простейшем) коде демонстрируются события и их обработчики в действии:

```

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
    SavingChanges += (sender, args) =>
    {
        Console.WriteLine($"Saving changes for {((DbContext)sender).
Database.GetConnectionString()}");
    };
    SavedChanges += (sender, args) =>
    {
        Console.WriteLine($"Saved {args.EntitiesSavedCount} entities");
    };
    SaveChangesFailed += (sender, args) =>
    {
        // Сгенерировано исключение.
        Console.WriteLine($"An exception occurred! {args.Exception.Message}");
    };
}

```

Класс DbSet<T>

Для каждой сущности в своей объектной модели вы добавляете свойство типа `DbSet<T>`. Класс `DbSet<T>` представляет собой специализированную коллекцию, используемую для взаимодействия с поставщиком баз данных с целью получения, добавления, обновления и удаления записей в базе данных. Каждая коллекция `DbSet<T>` предлагает несколько основных служб для взаимодействия с базой данных. Любые запросы LINQ, запускаемые в отношении класса `DbSet<T>`, транслируются поставщиком базы данных в запросы к базе данных. В табл. 22.2 описан ряд основных членов класса `DbSet<T>`.

Таблица 22.2. Распространенные члены DbSet<T>

Члены	Описание
Add () /AddRange ()	Начинают отслеживание сущности или сущностей в состоянии Added. Элемент или элементы будут добавлены в результате вызова SaveChanges (). Доступны также асинхронные версии
AsAsyncEnumerable ()	Возвращает коллекцию как реализацию IAsyncEnumerable<T>
AsQueryable ()	Возвращает экземпляр реализации IQueryable<T> из DbSet<T>
Find ()	Ищет сущность в ChangeTracker по первичному ключу. Если сущность не найдена в ChangeTracker, тогда она запрашивается в хранилище данных. Доступна также асинхронная версия
Update () /UpdateRange ()	Начинают отслеживание сущности или сущностей в состоянии Modified. Элемент или элементы будут обновлены в результате вызова SaveChanges (). Доступны также асинхронные версии
Remove () /RemoveRange ()	Начинают отслеживание сущности или сущностей в состоянии Deleted. Элемент или элементы будут удалены в результате вызова SaveChanges (). Доступны также асинхронные версии
Attach () /AttachRange ()	Начинают отслеживание сущности или сущностей. Сущности с числовыми первичными ключами определяются как идентичность, а нулевое значение отлеживается как Added. Все остальные значения отлеживаются как Unchanged. Доступны также асинхронные версии
FromSqlRaw () / FromSqlInterpolated ()	Создает запрос LINQ на основе низкоуровневой или интерполированной строки, представляющей запрос SQL. Может комбинироваться с дополнительными операторами LINQ для выполнения на стороне сервера

Класс DbSet<T> реализует интерфейс IQueryable<T> и обычно является целью запросов LINQ to Entity. Помимо расширяющих методов, добавленных инфраструктурой EF Core, класс DbSet<T> поддерживает расширяющие методы, которые вы изучили в главе 13, такие как ForEach(), Select() и All().

Вы узнаете, как добавлять к классу ApplicationContext свойства типа DbSet<T>, в разделе “Сущности” далее в главе.

На заметку! Многие методы из перечисленных в табл. 22.2, имеют те же самые имена, что и методы в табл. 22.1. Основное отличие в том, что методам DbSet<T> уже известен тип, с которым нужно работать, и список сущностей. Методы DbContext обязаны определять, на чем действовать, с применением рефлексии. Методы DbSet<T> используются гораздо чаще, чем методы DbContext.

Типы запросов

Типы запросов — это коллекции `DbSet<T>`, которые применяются для изображения представлений, оператора SQL или таблиц без первичного ключа. В предшествующих версиях EF Core для всего упомянутого использовался тип `DbQuery<T>`, но начиная с EF Core 3.1, тип `DbQuery` больше не употребляется. Типы запросов добавляются к производному классу `DbContext` с применением свойств `DbSet<T>` и конфигурируются как не имеющие ключей.

Например, класс `CustomerOrderViewModel` (который вы создадите при построении полной библиотеки доступа к данным `AutoLot`) конфигурируется с атрибутом `[Keyless]`:

```
[Keyless]
public class CustomerOrderViewModel
{
    ...
}
```

Остальные действия по конфигурированию делаются в Fluent API. В следующем примере сущность устанавливается как не имеющая ключа, а тип запроса сопоставляется с представлением базы данных `dbo.CustomerOrderView` (обратите внимание, что вызов метода `HasKey()` из Fluent API не требуется, если в модели присутствует аннотация данных `Keyless`, и наоборот, но он показан ради полноты):

```
modelBuilder.Entity<CustomerOrderViewModel>().HasNoKey()
    .ToView("CustomerOrderView", "dbo");
```

Типы запросов могут также сопоставляться с запросом SQL, как показано ниже:

```
modelBuilder.Entity<CustomerOrderViewModel>().HasNoKey()
    .ToSqlQuery(
        @"SELECT c.FirstName, c.LastName, i.Color, i.PetName, m.Name AS Make
        FROM dbo.Orders o
        INNER JOIN dbo.Customers c ON o.CustomerId = c.Id
        INNER JOIN dbo.Inventory i ON o.CarId = i.Id
        INNER JOIN dbo.Makes m ON m.Id = i.MakeId");
```

Последние механизмы, с которыми можно использовать типы запросов — это методы `FromSqlRaw()` и `FromSqlInterpolated()`. Вот пример того же самого запроса, но с применением `FromSqlRaw()`:

```
public IEnumerable<CustomerOrderViewModel> GetOrders()
{
    return CustomerOrderViewModels.FromSqlRaw(
        @"SELECT c.FirstName, c.LastName, i.Color, i.PetName, m.Name AS Make
        FROM dbo.Orders o
        INNER JOIN dbo.Customers c ON o.CustomerId = c.Id
        INNER JOIN dbo.Inventory i ON o.CarId = i.Id
        INNER JOIN dbo.Makes m ON m.Id = i.MakeId");
}
```

Гибкое сопоставление с запросом или таблицей

В версии EF Core 5 появилась возможность сопоставления одного и того же класса с более чем одним объектом базы данных. Такими объектами могут быть таблицы, представления или функции. Например, класс `CarViewModel` из главы 21 может

отображаться на представление, которое возвращает название производителя с данными Car и таблицей Inventory. Затем EF Core будет запрашивать из представления и отправлять обновления таблице:

```
modelBuilder.Entity<CarViewModel>()
    .ToTable("Inventory")
    .ToView("InventoryWithMakesView");
```

Экземпляр ChangeTracker

Экземпляр ChangeTracker отслеживает состояние объектов, загруженных в DbSet<T> внутри экземпляра DbContext. В табл. 22.3 описаны возможные значения для состояния объекта.

Таблица 22.3. Значения перечисления EntityState

Значение	Описание
Added	Сущность отслеживается, но пока не существует в базе данных
Deleted	Сущность отслеживается и помечена для удаления из базы данных
Detached	Сущность не отслеживается средством отслеживания изменений
Modified	Сущность отслеживается и была изменена
Unchanged	Сущность отслеживается, существует в базе данных и не была модифицирована

Для проверки состояния объекта используйте следующий код:

```
EntityState state = context.Entry(entity).State;
```

Вы также можете программно изменять состояние объекта с применением того же самого механизма. Чтобы изменить состояние на Deleted, используйте такой код:

```
context.Entry(entity).State = EntityState.Deleted;
```

События ChangeTracker

Экземпляр ChangeTracker способен генерировать два события: StateChanged и Tracked. Событие StateChanged инициируется в случае изменения состояния сущности. Оно не генерируется при отслеживании сущности в первый раз. Событие Tracked инициируется, когда сущность начинает отслеживаться, либо за счет добавления экземпляра DbSet<T> в коде, либо при возвращении из запроса.

Сброс состояния DbContext

В версии EF Core 5 появилась возможность сброса состояния DbContext. Метод ChangeTracker.Clear() отсоединяет все сущности от свойств DbSet<T>, устанавливая их состояние в Detached.

Сущности

Строго типизированные классы, которые сопоставляются с таблицами базы данных, официально именуются *сущностями*. Коллекция сущностей в приложении образует концептуальную модель физической базы данных. Выражаясь формально, такая модель называется *моделью сущностных данных* (entity data model — EDM) или прос-

то моделью. Модель сопоставляется с предметной областью приложения. Сущности и их свойства отображаются на таблицы и столбцы с применением соглашений Entity Framework Core, конфигурации и Fluent API (кода). Сущности не обязаны быть сопоставленными напрямую со схемой базы данных. Вы можете структурировать сущностные классы согласно потребностям создаваемого приложения и затем отобразить свои уникальные сущности на схему базы данных.

Подобная слабая связанность между базой данных и вашими сущностями означает возможность придания сущностям формы, соответствующей предметной области, независимо от проектного решения и структуры базы данных. Например, возьмем простую таблицу `Inventory` из базы данных `AutoLot` и сущностный класс `Car` из предыдущей главы. Имена отличаются, но сущность `Car` сопоставляется с таблицей `Inventory`. Исполняющая среда EF Core исследует конфигурацию сущностей в модели, чтобы отобразить клиентское представление таблицы `Inventory` (класс `Car` в примере) на корректные столбцы таблицы `Inventory`.

В последующих разделах будет показано, каким образом соглашения EF Core, аннотации данных и код (использующий Fluent API) сопоставляют сущности, свойства и отношения между сущностями в модели с таблицами, столбцами и отношениями внешних ключей в базе данных.

Сопоставление свойств со столбцами

При работе с реляционным хранилищем данных по соглашениям EF Core все открытые свойства, допускающие чтение и запись, сопоставляются со столбцами таблицы, на которую отображается сущность. Если свойство является автоматическим, то EF Core читает и записывает через методы получения и установки. Если свойство имеет поддерживаемое поле, тогда EF Core будет читать и записывать не в открытое свойство, а в поддерживаемое поле, хотя оно и закрыто. Несмотря на то что EF Core может читать и записывать в закрытые поля, все же должно быть предусмотрено открытое свойство, предназначенное для чтения и записи, которое инкапсулирует поддерживающее поле.

Наличие поддерживающих полей предпочтительнее в двух сценариях: при использовании шаблона `INotifyPropertyChanged` в приложениях Windows Presentation Foundation (WPF) и при возникновении конфликта между стандартными значениями базы данных и стандартными значениями .NET Core. Применение EF Core с WPF обсуждается в главе 28, а стандартные значения базы данных раскрываются позже в текущей главе.

Имена, типы данных и допустимость значений `null` столбцов конфигурируются через соглашения, аннотации данных и/или Fluent API. Все указанные темы подробно рассматриваются далее в главе.

Сопоставление классов с таблицами

В EF Core доступны две схемы сопоставления классов с таблицами: "таблица на иерархию" (`table-per-hierarchy` — TPH) и "таблица на тип" (`table-per-type` — TPT). Сопоставление TPH используется по умолчанию и отображает иерархию наследования на единственную таблицу. Появившееся в версии EF Core 5 сопоставление TPT отображает каждый класс в иерархии на собственную таблицу.

На заметку! Классы также можно отображать на представления и низкоуровневые запросы SQL. Они называются типами запросов и обсуждаются позже в главе.

Сопоставление “таблица на иерархию” (TPH)

Рассмотрим приведенный ниже пример, в котором класс Car из главы 21 разделен на два класса: базовый класс для свойств Id и TimeStamp и собственно класс Car с остальными свойствами. Оба класса должны быть созданы в папке Models проекта AutoLot.Samples:

```
using System.Collections.Generic;
namespace AutoLot.Samples.Models
{
    public abstract class BaseEntity
    {
        public int Id { get; set; }
        public byte[] TimeStamp { get; set; }
    }
}

using System.Collections.Generic;
namespace AutoLot.Samples.Models
{
    public class Car : BaseEntity
    {
        public string Color { get; set; }
        public string PetName { get; set; }
        public int MakeId { get; set; }
    }
}
```

Чтобы уведомить EF Core о том, что сущностный класс является частью объектной модели, предусмотрите свойство `DbSet<T>` для сущности. Добавьте в класс `ApplicationDbContext` такой оператор `using`:

```
using AutoLot.Samples.Models;
```

Поместите следующий код в класс `ApplicationDbContext` между конструктором и методом `OnModelCreating()`:

```
public DbSet<Car> Cars { get; set; }
```

Обратите внимание, что базовый класс не добавляется в виде экземпляра `DbSet<T>`. Хотя подробные сведения о миграциях приводятся позже в главе, давайте создадим базу данных и таблицу Cars. Откройте окно командной строки в папке проекта `AutoLot.Samples` и выполните показанные ниже команды:

```
dotnet tool install --global dotnet-ef --version 5.0.1
dotnet ef migrations add TPH -o Migrations
    -c AutoLot.Samples.ApplicationDbContext
dotnet ef database update TPH -c AutoLot.Samples.ApplicationDbContext
```

Первая команда устанавливает инструменты командной строки EF Core как глобальные. На вашей машине это понадобится сделать только раз. Вторая команда создает в папке `Migrations` миграцию по имени TPH с применением класса `ApplicationDbContext` в пространстве имен `AutoLot.Samples`. Третья команда обновляет базу на основе миграции TPH.

Когда EF Core используется для создания этой таблицы в базе данных, то унаследованный класс BaseEntity объединяется с классом Car и создается единственная таблица:

```
CREATE TABLE [dbo].[Cars] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [MakeId] [int] NOT NULL,
    [Color] [nvarchar](max) NULL,
    [PetName] [nvarchar](max) NULL,
    [TimeStamp] [varbinary](max) NULL,
    CONSTRAINT [PK_Cars] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
    OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

В предыдущем примере для создания свойств таблицы и столбцов применялись соглашения EF Core (раскрываемые вскоре).

Сопоставление “таблица на тип” (TPT)

Для изучения схемы сопоставления TPT можно использовать те же самые сущности, что и ранее, даже если базовый класс помечен как абстрактный. Поскольку схема TPH применяется по умолчанию, инфраструктуру EF Core необходимо проинструктировать для отображения каждого класса на таблицу, что можно сделать с помощью аннотаций данных или Fluent API. Добавьте в ApplicationDbContext следующий код:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<BaseEntity>().ToTable("BaseEntities");
    modelBuilder.Entity<Car>().ToTable("Cars");
    OnModelCreatingPartial(modelBuilder);
}
partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
```

Чтобы “сбросить” базу данных и проект, удалите папку Migrations и базу данных. Вот как удалить базу данных в CLI:

```
dotnet ef database drop -f -c AutoLot.Samples.ApplicationDbContext
```

Теперь создайте и примените миграцию для схемы TPT:

```
dotnet ef migrations add TPT -o Migrations
-c AutoLot.Samples.ApplicationDbContext
dotnet ef database update TPT -c AutoLot.Samples.ApplicationDbContext
```

При обновлении базы данных исполняющая среда EF Core создаст следующие таблицы. Индексы также показывают, что таблицы имеют сопоставление “один к одному”:

```
CREATE TABLE [dbo].[BaseEntities] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [TimeStamp] [varbinary](max) NULL,
    CONSTRAINT [PK_BaseEntities] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
```

```

)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
      IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
      OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

CREATE TABLE [dbo].[Inventory](
    [Id] [int] NOT NULL,
    [MakeId] [int] NOT NULL,
    [Color] [nvarchar](max) NULL,
    [PetName] [nvarchar](max) NULL,
CONSTRAINT [PK_Inventory] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
      IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
      OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

ALTER TABLE [dbo].[Inventory]
WITH CHECK ADD CONSTRAINT [FK_Inventory_BaseEntities_Id]
FOREIGN KEY([Id])
REFERENCES [dbo].[BaseEntities] ([Id])
GO
ALTER TABLE [dbo].[Inventory] CHECK CONSTRAINT [FK_Inventory_BaseEntities_Id]
GO

```

На заметку! С сопоставлением TPT связаны значительные последствия в плане производительности, которые должны приниматься во внимание при выборе данной схемы сопоставления. Дополнительные сведения изучите в документации: <https://docs.microsoft.com/ru-ru/ef/core/performance/modeling-for-performance#inheritance-mapping>.

Чтобы “сбросить” базу данных и проект для подготовки к следующему набору примеров, закомментируйте код в методе `OnModelCreating()` и опять удалите папку `Migrations` вместе с базой данных:

```
dotnet ef database drop -f -c AutoLot.Samples.ApplicationDbContext
```

Навигационные свойства и внешние ключи

Навигационные свойства представляют то, каким образом сущностные классы связаны между собой, и позволяют проходить от одного экземпляра сущности к другому. По определению навигационным является любое свойство, которое отображается на нескалярный тип, как определено поставщиком базы данных. На практике навигационное свойство сопоставляется с другой сущностью (*навигационное свойство типа ссылки*) или с коллекцией других сущностей (*навигационное свойство типа коллекций*). На стороне базы данных навигационные свойства транслируются в отношения внешнего ключа между таблицами. Инфраструктура EF Core напрямую поддерживает отношения вида “один к одному”, “один ко многим” и (в версии EF Core 5) “многие ко многим”. Сущностные классы также могут иметь обратные навигационные свойства с самими собой, представляя самоссылающиеся таблицы.

На заметку! Объекты с навигационными свойствами удобно рассматривать как связные списки. Если навигационные свойства являются двунаправленными, тогда объекты ведут себя подобно двусвязным спискам.

Прежде чем погружаться в детали навигационных свойств и шаблонов отношений между сущностями, ознакомьтесь с табл. 22.4, где приведены термины, которые используются во всем трех шаблонах отношений.

Таблица 22.4. Термины, применяемые для описания навигационных свойств и отношений

Термин	Описание
Главная (principal) сущность	Родительская часть отношения
Зависимая сущность	Дочерняя часть отношения
Главный ключ	Свойство или свойства, используемые для определения главной сущности. Может быть первичным или альтернативным ключом. Ключи могут быть сконфигурированы с применением одиночного свойства или множества свойств
Внешний ключ	Свойство или свойства, поддерживаемые дочерней сущностью для хранения главного ключа
Обязательное отношение	Отношение, в котором внешний ключ обязателен (не допускает значения null)
Необязательное отношение	Отношение, в котором внешний ключ не обязателен (допускает значения null)

Отсутствие свойств для внешних ключей

Если сущность с навигационным свойством типа ссылки не имеет свойства для значения внешнего ключа, тогда EF Core создаст необходимое свойство или свойства внутри сущности. Они известны как *теневые свойства внешних ключей* и именуются в формате <имя навигационного свойства><имя свойства главного ключа> или <имя главной сущности><имя свойства главного ключа>. Сказанное справедливо для всех видов отношений (“один ко многим”, “один к одному”, “многие ко многим”). Такой подход к построению сущностей с явным свойством или свойствами внешних ключей гораздо яснее, чем поручение их создания исполняющей среде EF Core.

Отношения “один ко многим”

Чтобы создать отношение “один ко многим”, сущностный класс со стороны “один” (главная сущность) добавляет свойство типа коллекции сущностных классов, находящихся на стороне “многие” (зависимые сущности). Зависимая сущность также должна иметь свойства для внешнего ключа обратно к главной сущности, иначе исполняющая среда EF Core создаст теневые свойства внешних ключей, как объяснялось ранее.

Например, в базе данных, созданной в главе 21, между таблицей Makes (представленной сущностным классом Make) и таблицей Inventory (представленной сущностным классом Car) имеется отношение “один ко многим”. Для упрощения примеров сущность Car будет отображаться на таблицу Cars. В следующем коде показаны двунаправленные навигационные свойства, представляющие это отношение:

```

using System.Collections.Generic;
namespace AutoLot.Samples.Models
{
    public class Make : BaseEntity
    {
        public string Name { get; set; }
        public I Enumerable<Car> Cars { get; set; } = new List<Car>();
    }
}
using System.Collections.Generic;
namespace AutoLot.Samples.Models
{
    public class Car : BaseEntity
    {
        public string Color { get; set; }
        public string PetName { get; set; }
        public int MakeId { get; set; }
        public Make MakeNavigation { get; set; }
    }
}

```

На заметку! При создании шаблонов для существующей базы данных исполняющая среда EF Core именует навигационные свойства типа ссылок аналогично обычным свойствам (скажем, `public Make {get; set;}`). В итоге могут возникать проблемы с навигацией и IntelliSense, не говоря уже о затруднениях при работе с кодом. Для ясности предпочтительнее добавлять к именам навигационных свойств типа ссылок суффикс `Navigation`, как демонстрировалось выше.

В примере `Car/Make` сущность `Car` является зависимой ("многие" в "один ко многим"), а сущность `Make` — главной ("один" в "один ко многим").

Добавьте в класс `ApplicationDbContext` экземпляр `DbSet<Make>`:

```

public DbSet<Car> Cars { get; set; }
public DbSet<Make> Makes { get; set; }

```

Затем создайте миграцию и обновите базу данных с использованием приведенных далее команд:

```

dotnet ef migrations add One2Many -o Migrations
-c AutoLot.Samples.ApplicationDbContext
dotnet ef database update One2Many -c AutoLot.Samples.ApplicationDbContext

```

При обновлении базы данных с применением миграции EF Core создаются следующие таблицы:

```

CREATE TABLE [dbo].[Makes] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](max) NULL,
    [TimeStamp] [varbinary](max) NULL,
    CONSTRAINT [PK_Makes] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    ) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
    OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]

```

```

GO

CREATE TABLE [dbo].[Cars] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [Color] [nvarchar](max) NULL,
    [PetName] [nvarchar](max) NULL,
    [TimeStamp] [varbinary](max) NULL,
    [MakeId] [int] NOT NULL,
    CONSTRAINT [PK_Cars] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
    OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

ALTER TABLE [dbo].[Cars]
WITH CHECK ADD CONSTRAINT [FK_Cars_Makes_MakeId] FOREIGN
KEY([MakeId])
REFERENCES [dbo].[Makes] ([Id])
ON DELETE CASCADE
GO

ALTER TABLE [dbo].[Cars] CHECK CONSTRAINT [FK_Cars_Makes_MakeId]
GO

```

Обратите внимание на ограничения внешнего ключа и проверки, созданные для зависимой таблицы (Cars).

Отношения “один к одному”

В отношениях “один к одному” обе сущности имеют навигационные свойства типа ссылок друг на друга. Хотя отношения “один к одному” четко обозначают главную и зависимую сущности, при построении таких отношений EF Core необходимо сообщить о том, какая сторона является главной, либо явно определяя внешний ключ, либо указывая главную сущность с использованием Fluent API. Если не проинформировать исполняющую среду EF Core, тогда она будет делать выбор на основе своей способности обнаруживать внешний ключ. На практике вы должны четко определять зависимую сущность, добавляя свойства внешнего ключа:

```

namespace AutoLot.Samples.Models
{
    public class Car : BaseEntity
    {
        public string Color { get; set; }
        public string PetName { get; set; }
        public int MakeId { get; set; }
        public Make MakeNavigation { get; set; }
        public Radio RadioNavigation { get; set; }
    }
}

namespace AutoLot.Samples.Models
{
    public class Radio : BaseEntity
    {
}

```

```

    public bool HasTweeters { get; set; }
    public bool HasSubWoofers { get; set; }
    public string RadioId { get; set; }
    public int CarId { get; set; }
    public Car CarNavigation { get; set; }
}
}

```

Поскольку класс Radio имеет внешний ключ к классу Car (на основе соглашения, которое будет раскрыто вскоре), Radio является зависимой, а Car — главной сущностью. Исполняющая среда EF Core неявно создает обязательный уникальный индекс на свойстве внешнего ключа в зависимой сущности. Если вы хотите изменить имя индекса, тогда можете воспользоваться аннотациями данных или Fluent API.

Добавьте в класс ApplicationDbContext экземпляр DbSet<Radio>:

```

public virtual DbSet<Car> Cars { get; set; }
public virtual DbSet<Make> Makes { get; set; }
public virtual DbSet<Radio> Radios { get; set; }

```

Создайте миграцию и обновите базу данных с помощью таких команд:

```

dotnet ef migrations add One2One -o Migrations
-c AutoLot.Samples.ApplicationDbContext
dotnet ef database update One2One -c AutoLot.Samples.ApplicationDbContext

```

В результате обновления базы данных с применением миграции EF Core таблица Cars не изменяется, но создается таблица Radios:

```

CREATE TABLE [dbo].[Radios] (
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [HasTweeters] [bit] NOT NULL,
    [HasSubWoofers] [bit] NOT NULL,
    [RadioId] [nvarchar](max) NULL,
    [TimeStamp] [varbinary](max) NULL,
    [CarId] [int] NOT NULL,
CONSTRAINT [PK_Radios] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
    OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

ALTER TABLE [dbo].[Radios]
WITH CHECK ADD CONSTRAINT [FK_Radios_Cars_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[Cars] ([Id])
ON DELETE CASCADE
GO

ALTER TABLE [dbo].[Radios] CHECK CONSTRAINT [FK_Radios_Cars_CarId]
GO

```

Обратите внимание на ограничения внешнего ключа и проверки, созданные для зависимой таблицы (Radios).

Отношения “многие ко многим” (нововведение в версии EF Core 5)

В отношении “многие ко многим” каждая сущность содержит навигационное свойство типа коллекции для другой сущности, что в хранилище данных реализуется с использованием таблицы соединения посреди двух сущностных таблиц. Такая таблица соединения именуется в соответствии с двумя таблицами в виде <Сущность1Сущность2>. Имя можно изменить в коде через Fluent API. Таблица соединения имеет отношения “один ко многим” с каждой сущностной таблицей:

```
namespace AutoLot.Samples.Models
{
    public class Car : BaseEntity
    {
        public string Color { get; set; }
        public string PetName { get; set; }
        public int MakeId { get; set; }
        public Make MakeNavigation { get; set; }
        public Radio RadioNavigation { get; set; }
        public IEnumerable<Driver> Drivers { get; set; } = new List<Driver>();
    }
}

namespace AutoLot.Samples.Models
{
    public class Driver : BaseEntity
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public IEnumerable<Car> Cars { get; set; } = new List<Car>();
    }
}
```

Эквивалентное решение можно обеспечить путем явного создания трех таблиц и именно так приходилось поступать в версиях EF Core, предшествующих EF Core 5. Вот как выглядит сокращенный пример:

```
public class Driver
{
    ...
    public IEnumerable<CarDriver> CarDrivers { get; set; }
}
public class Car
{
    ...
    public IEnumerable<CarDriver> CarDrivers { get; set; }
}
public class CarDriver
{
    public int CarId {get;set;}
    public Car CarNavigation {get;set;}
    public int DriverId {get;set;}
    public Driver DriverNavigation {get;set;}
}
```

Добавьте в класс ApplicationDbContext экземпляр DbSet<Driver>:

```
public virtual DbSet<Car> Cars { get; set; }
public virtual DbSet<Make> Makes { get; set; }
public virtual DbSet<Radio> Radios { get; set; }
public virtual DbSet<Driver> Drivers { get; set; }
```

Создайте миграцию и обновите базу данных с помощью следующих команд:

```
dotnet ef migrations add Many2Many -o Migrations
-c AutoLot.Samples.ApplicationDbContext
dotnet ef database update many2Many
-c AutoLot.Samples.ApplicationDbContext
```

После обновления базы данных с применением миграции EF Core таблица Cars не изменяется, но создаются таблицы Drivers и CarDriver:

```
CREATE TABLE [dbo].[Drivers](
    [Id] [INT] IDENTITY(1,1) NOT NULL,
    [FirstName] [NVARCHAR](MAX) NULL,
    [LastName] [NVARCHAR](MAX) NULL,
    [TimeStamp] [VARBINARY](MAX) NULL,
    CONSTRAINT [PK_Drivers] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
    ALLOW_PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
CREATE TABLE [dbo].[CarDriver](
    [CarsId] [int] NOT NULL,
    [DriversId] [int] NOT NULL,
    CONSTRAINT [PK_CarDriver] PRIMARY KEY CLUSTERED
    (
        [CarsId] ASC,
        [DriversId] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
    OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[CarDriver]
WITH CHECK ADD CONSTRAINT [FK_CarDriver_Cars_CarsId] FOREIGN
KEY([CarsId])
REFERENCES [dbo].[Cars] ([Id])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[CarDriver] CHECK CONSTRAINT [FK_CarDriver_Cars_CarsId]
GO
ALTER TABLE [dbo].[CarDriver]
WITH CHECK ADD CONSTRAINT [FK_CarDriver_Drivers_DriversId]
FOREIGN KEY([DriversId])
REFERENCES [dbo].[Drivers] ([Id])
ON DELETE CASCADE
GO
ALTER TABLE [dbo].[CarDriver]
CHECK CONSTRAINT [FK_CarDriver_Drivers_DriversId]
GO
```

Обратите внимание на то, что исполняющая среда EF Core создает составной первичный ключ, ограничения проверки (внешних ключей) и каскадное поведение, чтобы обеспечить конфигурирование таблицы CarDriver как надлежащей таблицы соединения.

На заметку! На момент написания главы создание шаблонов для отношений “многие ко многим” пока не поддерживалось. Создание шаблонов для отношений “многие ко многим” основано на табличной структуре, как во втором примере сущностью CarDriver. Дополнительные сведения о проблеме доступны по ссылке <https://github.com/dotnet/efcore/issues/22475>.

Каскадное поведение

В большинстве хранилищ данных (вроде SQL Server) установлены правила, управляющие поведением при удалении строки. Если связанные (зависимые) записи тоже должны быть удалены, то такой подход называется **каскадным удалением**. В EF Core существуют три действия, которые могут произойти при удалении главной сущности (с зависимыми сущностями, загруженными в память):

- зависимые записи удаляются;
- зависимые внешние ключи устанавливаются в null;
- зависимые сущности остаются незатронутыми.

Стандартное поведение для необязательных и обязательных отношений отличается. Поведение можно установить в одно из семи значений, из которых рекомендуется использовать только пять. Поведение конфигурируется с применением перечисления `DeleteBehavior` посредством Fluent API. Ниже перечислены доступные варианты в перечислении:

- Cascade;
- ClientCascade;
- ClientNoAction (не рекомендуется к использованию);
- ClientSetNull;
- NoAction (не рекомендуется к использованию);
- SetNull;
- Restrict.

Указанное поведение в EF Core инициируется только после удаления сущности и вызова метода `SaveChanges()` на экземпляре класса, унаследованного от `DbContext`. Дополнительные сведения о том, когда EF Core взаимодействует с хранилищем данных, ищите в разделе “Выполнение запросов” далее в главе.

Необязательные отношения

Вспомните из табл. 22.4, что необязательными отношениями считаются такие, в которых зависимая сущность может устанавливать значение или значения внешних ключей в null. Для необязательных отношений стандартным поведением является `ClientSetNull`. В табл. 22.5 описано каскадное поведение с зависимыми сущностями и влияние на записи базы данных при использовании SQL Server.

Таблица 22.5. Каскадное поведение для необязательных отношений

Поведение удаления	Влияние на зависимые сущности (в памяти)	Влияние на зависимые сущности (в базе данных)
Cascade	Сущности удаляются	Сущности удаляются базой данных
ClientCascade	Сущности удаляются	Для баз данных, которые не поддерживают каскадное удаление, сущности удаляются исполняющей средой EF Core
ClientSetNull (по умолчанию)	Свойство или свойства внешних ключей устанавливаются в null	–
SetNull	Свойство или свойства внешних ключей устанавливаются в null	Свойство или свойства внешних ключей устанавливаются в null
Restrict	–	–

Обязательные отношения

Обязательные отношения — это такие отношения, при которых зависимая сущность *не может* устанавливать значение или значения внешних ключей в null. Для обязательных отношений стандартным поведением является Cascade. В табл. 22.6 описано каскадное поведение с зависимыми сущностями и влияние на записи базы данных при использовании SQL Server.

Таблица 22.6. Каскадное поведение для обязательных отношений

Поведение удаления	Влияние на зависимые сущности (в памяти)	Влияние на зависимые сущности (в базе данных)
Cascade (по умолчанию)	Сущности удаляются	Сущности удаляются
ClientCascade	Сущности удаляются	Для баз данных, которые не поддерживают каскадное удаление, сущности удаляются исполняющей средой EF Core
ClientSetNull	Метод SaveChanges () генерирует исключение	–
SetNull	Метод SaveChanges () генерирует исключение	Метод SaveChanges () генерирует исключение
Restrict	–	–

Соглашения, связанные с сущностями

В EF Core принято много соглашений для определения сущности и ее связи с хранилищем данных. Соглашения всегда включены, если только они не отменены аннотациями данных или кодом Fluent API. В табл. 22.7 перечислены наиболее важные соглашения EF Core.

Таблица 22.7. Наиболее важные соглашения EF Core

Соглашение	Описание
Включаемые таблицы	В базе данных создаются все классы со свойством DbSet и все классы, которые достижимы (через навигационные свойства) классом DbSet
Включаемые столбцы	На столбцы отображаются все открытые свойства с методами получения и установки (включая автоматические свойства)
Имя таблицы	Отображается на имя свойства DbSet в классе, производном от DbContext. Если свойств DbSet не существует, тогда используется имя класса
Схема	Таблицы создаются в стандартной схеме хранилища данных (dbo в SQL Server)
Имя столбца	Имена столбцов отображаются на имена свойств класса
Тип данных столбца	Типы данных выбираются на основе типов данных .NET Core и транслируются поставщиком базы данных (SQL Server). Тип данных DateTime отображается на datetime2(7), а string — на nvarchar(max). Строки, являющиеся частью первичного ключа, отображаются на nvarchar(450)
Допустимость значения null в столбце	Типы данных, не допускающие значение null, создаются как постоянные столбцы Not Null. В EF Core соблюдается допустимость значения null из версии C# 8
Первичный ключ	Свойства с именами Id или <ИмяСущностногоТипа>Id будут конфигурироваться как первичный ключ. Ключи типа short, int, long или Guid имеют значения, управляемые хранилищем данных. Числовые значения создаются в виде столбцов Identity (SQL Server)
Отношения	Отношения между таблицами создаются при наличии навигационных свойств между сущностными классами
Внешний ключ	Свойства с именами <ИмяДругогоКласса>Id являются внешними ключами для навигационных свойств типа <ИмяДругогоКласса>

Во всех предшествующих примерах навигационных свойств для построения отношений между таблицами были задействованы соглашения EF Core.

Отображение свойств на столбцы

По соглашению открытые свойства для чтения и записи отображаются на столбцы с теми же самыми именами. Типы данных столбцов соответствуют эквивалентам для типов данных CLR свойств, принятым в хранилище данных. Свойства, не допускающие null, устанавливаются в хранилище данных как не null, а свойства, допускающие null, устанавливаются так, чтобы значение null было разрешено. Инфраструктура EF Core поддерживает ссылочные типы, допускающие null, которые появились в C# 8. Для поддерживающих полей EF Core ожидает их именования с применением одного из следующих соглашений (в порядке старшинства):

- _<имя свойства в "верблюжьем" стиле>
- _<имя свойства>
- m_<имя свойства в "верблюжьем" стиле>
- m_<имя свойства>

В случае обновления свойства Color класса Car для использования поддерживающего поля (по соглашению) оно получило бы имя _color, _Color, m_color или m_Color, как показано ниже:

```

private string _color = "Gold";
public string Color
{
    get => _color;
    set => _color = value;
}

```

Аннотации данных Entity Framework

Аннотации данных — это атрибуты C#, которые применяются для дальнейшего придания формы вашим сущностям. В табл. 22.8 описаны самые часто используемые аннотации данных, предназначенные для определения деталей того, как ваши сущностные классы и свойства сопоставляются с таблицами и полями базы данных. Аннотации данных переопределяют любые конфликтующие соглашения. В оставшемся материале главы и книги вы увидите еще много аннотаций, которые можно применять для уточнения сущностей в модели.

Таблица 22.8. Часто используемые аннотации данных, поддерживаемые Entity Framework Core (с помощью * помечены атрибуты, появившиеся в EF Core 5)

Аннотация данных	Описание
Table	Определяет имя схемы и таблицы для сущности
Keyless*	Указывает, что сущность не имеет ключа (например, является представлением базы данных)
Column	Определяет имя столбца для свойства сущности
BackingField*	Указывает поддерживающее поле C# для свойства
Key	Определяет первичный ключ для сущности. Поля ключа также неявно являются [Required]
Index*	Размещается в классе для указания индекса из одного или нескольких столбцов. Позволяет указывать, что индекс уникален
Owned	Объявляет, что классом будет владеть другой сущностный класс
Required	Объявляет свойство как не допускающее значение null в базе данных
ForeignKey	Объявляет свойство, которое применяется в качестве внешнего ключа для навигационного свойства
InverseProperty	Объявляет навигационное свойство на другом конце отношения
StringLength	Указывает максимальную длину для строкового свойства
TimeStamp	Объявляет тип как rowversion в SQL Server и добавляет проверки параллелизма к операциям базы данных, в которые вовлечена сущность
ConcurrencyCheck	Поле флагов, предназначенное для использования во время проверки параллелизма при выполнении обновлений и удалений
DatabaseGenerated	Указывает, сгенерировано ли поле базой данных. Принимает одно из значений перечисления DatabaseGeneratedOption — Computed, Identity или None
DataType	Обеспечивает более специфичное определение поля, чем внутренний тип данных
NotMapped	Исключает свойство или класс относительно полей и таблиц базы данных

В следующем коде показан класс `BaseEntity` с аннотацией, которая объявляет поле `Id` первичным ключом. Вторая аннотация свойства `Id` указывает, что оно является столбцом `Identity` в базе данных SQL Server. Свойство `TimeStamp` в SQL Server будет столбцом `timestamp/rowversion` (для проверки параллелизма, рассматриваемой позже в главе).

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
public abstract class BaseEntity
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    [Timestamp]
    public byte[] TimeStamp { get; set; }
}
```

Вот класс `Car` и аннотации данных, которые придают ему форму в базе данных:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;
[Table("Inventory", Schema="dbo")]
[Index(nameof(MakeId), Name = "IX_Inventory_MakeId")]
public class Car : BaseEntity
{
    [Required, StringLength(50)]
    public string Color { get; set; }
    [Required, StringLength(50)]
    public string PetName { get; set; }
    public int MakeId { get; set; }
    [ForeignKey(nameof(MakeId))]
    public Make MakeNavigation { get; set; }
    [InverseProperty(nameof(Driver.Cars))]
    public IEnumerable<Driver> Drivers { get; set; }
}
```

Атрибут `[Table]` сопоставляет класс `Car` с таблицей `Inventory` в схеме `dbo` (атрибут `[Column]` применяется для изменения имени столбца или типа данных). Атрибут `[Index]` создает индекс на внешнем ключе `MakeId`. Два строковых поля установлены как `[Required]` и имеющие максимальную длину (`StringLength`) в 50 символов. Атрибуты `[InverseProperty]` и `[ForeignKey]` объясняются в следующем разделе.

Ниже перечислены отличия от соглашений EF Core:

- переименование таблицы из `Cars` в `Inventory`;
- изменение типа данных столбца `TimeStamp` из `varbinary(max)` на `timestamp` в SQL Server;
- установка типа данных и допустимости значения `null` для столбцов `Color` и `PetName` вместо `nvarchar(max)/null` в `nvarchar(50)/ne null`;
- переименование индекса в `MakeId`.

Остальные используемые аннотации соответствуют конфигурации, определенной соглашениями EF Core.

Если вы создадите миграцию и попробуете ее применить, то миграция потерпит неудачу. СУБД SQL Server не разрешает изменять любой тип данных существующего столбца на timestamp. Столбец должен быть удален и затем воссоздан. К сожалению, инфраструктура миграций не позволяет удалять и воссоздавать, а пытается изменить столбец.

Вот как проще всего решить проблему: поместить свойство TimeStamp в комментарий внутри базовой сущности, создать и применить миграцию, убрать комментарий со свойства TimeStamp и затем создать и применить еще одну миграцию.

Закомментируйте свойство TimeStamp вместе с аннотацией данных и выполните следующие команды:

```
dotnet ef migrations add RemoveTimeStamp -o Migrations
  -c AutoLot.Samples.ApplicationDbContext
dotnet ef database update RemoveTimeStamp
  -c AutoLot.Samples.ApplicationDbContext
```

Уберите комментарий со свойства TimeStamp и аннотации данных и выполните показанные далее команды, чтобы добавить свойство TimeStamp в таблицу как столбец timestamp:

```
dotnet ef migrations add ReplaceTimeStamp -o Migrations
  -c AutoLot.Samples.ApplicationDbContext
dotnet ef database update ReplaceTimeStamp
  -c AutoLot.Samples.ApplicationDbContext
```

Теперь база данных соответствует вашей модели.

Аннотации и навигационные свойства

Аннотация ForeignKey позволяет EF Core знать, какое свойство является поддерживающим полем для навигационного свойства. По соглашению <ИмяТипа>Id автоматически станет свойством внешнего ключа, но в предыдущем примере оно было установлено явно. Такой подход обеспечивает отличающиеся стили именования, а также наличие в таблице более одного внешнего ключа. Кроме того, улучшается читабельность кода.

Свойство InverseProperty информирует EF Core о способе связывания таблиц, указывая навигационное свойство в других сущностях, которое направляет обратно в текущую сущность. Свойство InverseProperty требуется, когда сущность связана с другой сущностью несколько раз, и вдобавок делает код более читабельным.

Интерфейс Fluent API

С помощью интерфейса Fluent API сущности приложения конфигурируются посредством кода C#. Методы предоставляются экземпляром ModelBuilder, доступным в методе OnModelCreating() класса DbContext. Интерфейс Fluent API является самым мощным способом конфигурирования и переопределяет любые конфликтующие между собой соглашения или аннотации данных. Некоторые конфигурационные параметры доступны только через Fluent API, скажем, стандартные значения для настроек и каскадное поведение для навигационных свойств.

Отображение классов и свойств

В следующем коде воспроизведен предыдущий пример Car с использованием Fluent API вместо аннотаций данных (здесь не показаны навигационные свойства, которые будут раскрыты ниже):

154 Часть VII. Entity Framework Core

```
modelBuilder.Entity<Car>(entity =>
{
    entity.ToTable("Inventory", "dbo");
    entity.HasKey(e => e.Id);
    entity.HasIndex(e => e.MakeId, "IX_Inventory_MakeId");
    entity.Property(e => e.Color)
        .IsRequired()
        .HasMaxLength(50);
    entity.Property(e => e.PetName)
        .IsRequired()
        .HasMaxLength(50);
    entity.Property(e => e.TimeStamp)
        .IsRowVersion()
        .IsConcurrencyToken();
});
```

Если создать и запустить миграцию прямо сейчас, то вы обнаружите, что ничего не изменилось, поскольку вызываемые методы Fluent API соответствуют текущей конфигурации, определенной соглашениями и аннотациями данных.

Стандартные значения

Интерфейс Fluent API предлагает методы, позволяющие устанавливать стандартные значения для столбцов. Стандартное значение может иметь тип значения или быть строкой SQL. Например, вот как установить стандартное значение Color для новой сущности Car в Black:

```
modelBuilder.Entity<Car>(entity =>
{
    ...
    entity.Property(e => e.Color)
        .HasColumnName("CarColor")
        .IsRequired()
        .HasMaxLength(50)
        .HasDefaultValue("Black");
});
```

Чтобы установить значение для функции базы данных (вроде getdate()), примените метод HasDefaultValueSql(). Предположим, что в класс Car было добавлено свойство DateTime по имени DateBuilt, а стандартным значением должна быть текущая дата, получаемая с использованием метода getdate() в SQL Server. Столбцы конфигурируются следующим образом:

```
modelBuilder.Entity<Car>(entity =>
{
    ...
    entity.Property(e => e.DateBuilt)
        .HasDefaultValueSql("getdate()");
});
```

Как и в случае применения SQL для вставки записи, если свойство, которое отображается на столбец со стандартным значением, имеет значение, когда EF Core вставляет запись, то вместо стандартного значения столбца будет использоваться значение свойства. Если значение свойства равно null, тогда применяется стандартное значение столбца.

Проблема возникает при наличии стандартного значения у типа данных свойства. Вспомните, что стандартное значение для числовых типов составляет 0, а для булевских — false. Если вы установите значение числового свойства в 0 или булевского свойства в false и затем вставите такую сущность, тогда инфраструктура EF Core будет трактовать это свойство как *не имеющее установленного значения*. При отображении свойства на столбец со стандартным значением используется стандартное значение в определении столбца.

Например, добавьте в класс Car свойство типа bool по имени IsDriveable. Установите в true стандартное значение для отображения свойства на столбец:

```
// Car.cs
public class Car : BaseEntity
{
    ...
    public bool IsDriveable { get; set; }
}

// ApplicationDbContext
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>(entity =>
    {
        ...
        entity.Property(e => e.IsDriveable).HasDefaultValue(true);
    });
}
```

В случае если вы сохраните новую запись с IsDriveable = false, то значение false игнорируется (т.к. оно является стандартным значением для булевского типа) и будет применяться стандартное значение. Таким образом, значение для IsDriveable всегда будет равно true! Одно из решений предусматривает превращение вашего открытого свойства (и, следовательно, столбца) в допускающее null, но это может не соответствовать бизнес-потребностям.

Другое решение предлагается инфраструктурой EF Core, в частности, ее работой с поддерживающими полями. Вспомните, что если поддерживающее поле существует (и идентифицируется как такое для свойства через соглашение, аннотацию данных или Fluent API), тогда для действий по чтению и записи EF Core будет использовать поддерживающее поле, а не открытое свойство.

Если вы модифицируете IsDriveable с целью применения поддерживающего поля, допускающего null (но оставите свойство не допускающим null), то EF Core будет выполнять чтение и запись, используя поддерживающее поле, а не свойство. Стандартным значением для булевского типа, допускающего null, является null — не false. Описанное изменение обеспечит ожидаемое поведение свойства:

```
public class Car
{
    ...
    private bool? _isDriveable;
    public bool IsDriveable
    {
        get => _isDriveable ?? true;
        set => _isDriveable = value;
    }
}
```

Для информирования EF Core о поддерживающем поле используется Fluent API:

```
modelBuilder.Entity<Car>(entity =>
{
    entity.Property(p => p.IsDriveable)
        .HasField("_isDriveable")
        .HasDefaultValue(true);
});
```

На заметку! В приведенном примере вызов метода `HasField()` не обязательен, потому что имя поддерживающего поля следует соглашениям об именовании. Он включен в целях демонстрации применения Fluent API для указания поддерживающего поля.

Исполняющая среда EF Core транслирует поле в показанное ниже определение SQL:

```
CREATE TABLE [dbo].[Inventory] (
    ...
    [IsDriveable] [BIT] NOT NULL,
    ...
GO
ALTER TABLE [dbo].[Inventory] ADD DEFAULT (CONVERT([BIT],(1)))
FOR [IsDriveable]
GO
```

Вычисляемые столбцы

Столбцы также могут вычисляться на основе возможностей хранилища данных. Для SQL Server есть два варианта: вычислять значение, основываясь на других полях в той же самой записи, либо использовать скалярную функцию. Скажем, чтобы создать в таблице `Inventory` вычисляемый столбец, который объединяет значения `PetName` и `Color` для создания `DisplayName`, применяйте функцию `HasComputedColumnSql()`:

```
modelBuilder.Entity<Car>(entity =>
{
    entity.Property(p => p.FullName)
        .HasComputedColumnSql("[PetName] + ' (' + [Color] + ')'");
});
```

В версии EF Core 5 появилась возможность сохранения вычисляемых значений, так что значение вычисляется только при создании или обновлении строки. Хотя в SQL Server упомянутая возможность поддерживается, она присутствует не во всех хранилищах данных, поэтому проверяйте документацию по своему поставщику баз данных:

```
modelBuilder.Entity<Car>(entity =>
{
    entity.Property(p => p.FullName)
        .HasComputedColumnSql("[PetName] + ' (' + [Color] + ')'", 
        stored:true);
});
```

Отношения “один ко многим”

Чтобы определить отношение “один ко многим” с помощью Fluent API, выберите одну из сущностей, подлежащих обновлению. Обе стороны навигационной цепочки устанавливаются в одном блоке кода:

```
modelBuilder.Entity<Car>(entity =>
{
    ...
    entity.HasOne(d => d.MakeNavigation)
        .WithMany(p => p.Cars)
        .HasForeignKey(d => d.MakeId)
        .onDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_Inventory_Makes_MakeId");
});
```

Если вы выберете в качестве основы для конфигурации навигационной сущности главную сущность, тогда код будет выглядеть примерно так:

```
modelBuilder.Entity<Make>(entity =>
{
    ...
    entity.HasMany(e=>e.Cars)
        .WithOne(c=>c.MakeNavigation)
        .HasForeignKey(c=>c.MakeId)
        .onDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_Inventory_Makes_MakeId");
});
```

Отношения “один к одному”

Отношения “один к одному” конфигурируются аналогично, но только вместо метода `WithMany()` интерфейса Fluent API используется метод `WithOne()`. К зависимой сущности добавляется уникальный индекс. Вот код для отношения между сущностями `Car` и `Radio`, где применяется зависимая сущность (`Radio`):

```
modelBuilder.Entity<Radio>(entity =>
{
    entity.HasIndex(e => e.CarId, "IX_Radios_CarId")
        .IsUnique();
    entity.HasOne(d => d.CarNavigation)
        .WithOne(p => p.RadioNavigation)
        .HasForeignKey<Radio>(d => d.CarId);
});
```

Даже если отношение определено в главной сущности, то к зависимой сущности все равно добавляется уникальный индекс. Далее приведен код установки отношения между сущностями `Car` и `Radio`, где для отношения используется главная сущность:

```
modelBuilder.Entity<Radio>(entity =>
{
    entity.HasIndex(e => e.CarId, "IX_Radios_CarId")
        .IsUnique();
});

modelBuilder.Entity<Car>(entity =>
{
    entity.HasOne(d => d.RadioNavigation)
        .WithOne(p => p.CarNavigation)
        .HasForeignKey<Radio>(d => d.CarId);
});
```

Отношения “многие ко многим”

Отношения “многие ко многим” гораздо легче настраивать посредством Fluent API. Имена полей внешних ключей, имена индексов и каскадное поведение могут быть установлены в операторах, определяющих отношение. Ниже показан пример отношения “многие ко многим”, переделанный с применением Fluent API (имена ключей и столбцов были изменены, чтобы улучшить читабельность):

```
modelBuilder.Entity<Car>()
    .HasMany(p => p.Drivers)
    .WithMany(p => p.Cars)
    .UsingEntity<Dictionary<string, object>>(
        "CarDriver",
        j => j
            .HasOne<Driver>()
            .WithMany()
            .HasForeignKey("DriverId")
            .HasConstraintName("FK_CarDriver_Drivers_DriverId")
            .OnDelete(DeleteBehavior.Cascade),
        j => j
            .HasOne<Car>()
            .WithMany()
            .HasForeignKey("CarId")
            .HasConstraintName("FK_CarDriver_Cars_CarId")
            .OnDelete(DeleteBehavior.ClientCascade));
```

Соглашения, аннотации данных и Fluent API – что выбрать?

В настоящий момент вас может интересовать, какой из вариантов следует выбирать для формирования ваших сущностей, а также их связей друг с другом и с хранилищем данных? Ответ: все три. Соглашения активны всегда (если только вы не переопределите их посредством аннотаций данных или Fluent API). С помощью аннотаций данных можно делать почти все то, на что способны методы Fluent API, и хранить информацию в самом сущностном классе, повышая в ряде случаев читабельность кода и удобство его сопровождения. Из трех вариантов наиболее мощным является Fluent API, но код скрыт в классе DbContext. Независимо от того, используете вы аннотации данных или Fluent API, имейте в виду, что аннотации данных переопределяют встроенные соглашения, а методы Fluent API переопределяют вообще все.

Выполнение запросов

Запросы на извлечение данных создаются посредством запросов LINQ в отношении свойств DbSet<T>. На стороне сервера механизм трансляции LINQ поставщика баз данных видоизменяет запрос LINQ с учетом специфичного для базы данных языка (скажем, T-SQL). Запросы LINQ, охватывающие (или потенциально охватывающие) множество записей, не выполняются до тех пор, пока не начнется проход по результатам запросов (например, с применением foreach) или не произойдет привязка к элементу управления для их отображения (наподобие визуальной сетки данных). Такое отложенное выполнение позволяет строить запросы в коде, не испытывая проблем с производительностью из-за частого взаимодействия с базой данных.

Скажем, чтобы извлечь из базы данных все записи об автомобилях желтого цвета, запустите следующий запрос:

```
var cars = Context.Cars.Where(x=>x.Color == "Yellow");
```

Благодаря отложенному выполнению база данных фактически не запрашивается до тех пор, пока не начнется проход по результатам. Чтобы выполнить запрос немедленно, используйте `ToList()`:

```
var cars = Context.Cars.Where(x=>x.Color == "Yellow").ToList();
```

Поскольку запросы не выполняются до их запуска, их можно строить в нескольких строках кода. Показанный ниже пример кода делает то же самое, что и предыдущий пример:

```
var query = Context.Cars.AsQueryable();
query = query.Where(x=>x.Color == "Yellow");
var cars = query.ToList();
```

Запросы с одной записью (как в случае применения `First()`/`FirstOrDefault()`) выполняются немедленно при вызове действия (такого как `FirstOrDefault()`), а операторы создания, обновления и удаления выполняются немедленно, когда запускается метод `DbContext.SaveChanges()`.

Смешанное выполнение на клиентской и серверной сторонах

В предшествующих версиях EF Core была введена возможность смешивания выполнения на стороне сервера и на стороне клиента. Это означало, что где-то в середине оператора LINQ можно было бы вызвать функцию C# и по существу свести на нет все преимущества, описанные в предыдущем разделе. Часть до вызова функции C# выполнится на стороне сервера, но затем все результаты (в данной точке запроса) доставляются на сторону клиента и остаток запроса будет выполнен как LINQ to Objects. В итоге возможность смешанного выполнения привнесла больше проблем, нежели решила, и в выпуске EF Core 3.1 такая функциональность была изменена. Теперь выполнять на стороне клиента можно только последний узел оператора LINQ.

Сравнение отслеживаемых и неотслеживаемых запросов

При чтении информации из базы данных в экземпляр `DbSet<T>` сущности (по умолчанию) отслеживаются компонентом `ChangeTracker`, что обычно и требуется в приложении. Как только начинается отслеживание экземпляра компонентом `ChangeTracker`, любые последующие обращения к базе данных за тем же самым элементом (на основе первичного ключа) будут приводить к обновлению элемента, а не к его дублированию.

Однако временами из базы данных необходимо получать данные, отслеживать которые с помощью `ChangeTracker` нежелательно. Причина может быть связана с производительностью (отслеживание первоначальных и текущих значений для крупных наборов записей увеличивает нагрузку на память) либо же с тем фактом, что извлекаемые записи никогда не будут изменяться частью приложения, которая нуждается в этих данных.

Чтобы загрузить экземпляр `DbSet<T>`, не помещая данные в `ChangeTracker`, добавьте к оператору LINQ вызов `AsNoTracking()`, который указывает EF Core о необходимости извлечения данных без их помещения в `ChangeTracker`. Например, для загрузки записи `Car` без ее добавления в `ChangeTracker` введите следующий код:

```
public virtual Car? FindAsNoTracking(int id)
    => Table.AsNoTracking().FirstOrDefault(x => x.Id == id);
```

Преимущество показанного кода в том, что он не увеличивает нагрузку на память, но с ним связан и недостаток: дополнительные вызовы для извлечения той же самой записи Car создадут ее добавочные копии. За счет потребления большего объема памяти и чуть более длительного выполнения запрос можно модифицировать, чтобы гарантировать наличие только одного экземпляра несопоставленной сущности Car:

```
public virtual Car? FindAsNoTracking(int id)
    => Table.AsNoTrackingWithIdentityResolution().FirstOrDefault(x
        => x.Id == id);
```

Важные функциональные средства EF Core

Многие функциональные средства из EF 6 были воспроизведены в EF Core, а с каждым выпуском добавляются новые возможности. Множество средств в EF Core усовершенствовано как с точки зрения функциональности, так и в плане производительности. В дополнение к средствам, воспроизведенным из EF 6, инфраструктура EF Core располагает многочисленными новыми возможностями, которые в предыдущей версии отсутствовали. Ниже приведены наиболее важные функциональные средства инфраструктуры EF Core (в произвольном порядке).

На заметку! Фрагменты кода в текущем разделе взяты прямо из завершенной библиотеки доступа к данным AutoLot, которая будет построена в следующей главе.

Обработка значений, генерируемых базой данных

Помимо отслеживания изменений и генерации запросов SQL из LINQ существенным преимуществом использования EF Core по сравнению с низкоуровневой инфраструктурой ADO.NET является гладкая обработка значений, генерируемых базой данных. После добавления или обновления сущности исполняющая среда EF Core запрашивает любые данные, генерируемые базой, и автоматически обновляет сущность с применением корректных значений. При работе с низкоуровневой инфраструктурой ADO.NET это пришлось бы делать самостоятельно.

Например, таблица `Inventory` имеет целочисленный первичный ключ, который определяется в SQL Server как столбец `Identity`. Столбцы `Identity` заполняются СУБД SQL Server уникальными числами (из последовательности) при добавлении записи и не могут обновляться во время обычных обновлений (исключая особый случай `IDENTITY_INSERT`). Кроме того, таблица `Inventory` содержит столбец `Timestamp` для проверки параллелизма. Проверка параллелизма рассматривается далее, а пока достаточно знать, что столбец `Timestamp` поддерживается SQL Server и обновляется при любом действии добавления или редактирования.

В качестве примера возьмем добавление новой записи `Car` в таблицу `Inventory`. В приведенном ниже коде создается новый экземпляр `Car`, который добавляется к экземпляру `DbSet<Car>` класса, производного от `DbContext`, и вызывается метод `SaveChanges()` для сохранения данных:

```
var car = new Car
{
    Color = "Yellow",
    MakeId = 1,
    PetName = "Herbie"
};
```

```
Context.Cars.Add(car);
Context.SaveChanges();
```

При выполнении метода `SaveChanges()` в таблицу вставляется новая запись, после чего исполняющей среде EF Core возвращаются значения `Id` и `Timestamp` из таблицы, причем свойства сущности обновляются надлежащим образом:

```
INSERT INTO [Dbo].[Inventory] ([Color], [MakeId], [PetName])
VALUES (N'Yellow', 1, N'Herbie');
SELECT [Id], [TimeStamp]
FROM [Dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

На заметку! Фактически EF Core выполняет параметризованные запросы, но приводимые примеры упрощены ради читабельности.

Поступать так можно и при добавлении в базу данных множества элементов. Исполняющей среде EF Core известно, каким образом связывать значения с корректными сущностями. Когда записи обновляются, то значения первичных ключей уже известны, так что в нашем примере с `Car` запрашивается и возвращается только значение `Timestamp`.

Проверка параллелизма

Проблемы с параллелизмом возникают, когда два отдельных процесса (пользователя или системы) пытаются почти одновременно обновить ту же самую запись. Скажем, пользователи User 1 и User 2 получают данные для Customer A. Пользователь User 1 обновляет адрес и сохраняет изменения. Пользователь User 2 обновляет кредитный риск и пытается сохранить ту же запись. Если сохранение для пользователя User 2 сработало, тогда изменения от пользователя User 1 будут отменены, т.к. после того, как пользователь User 2 извлек запись, адрес изменился. Другой вариант — отказ сохранения для пользователя User 2, когда изменения для User 1 записываются, но изменения для User 2 — нет.

Обработка описанной ситуации зависит от требований приложения. Решения простираются от бездействия (второе обновление переписывает первое) и применения оптимистического параллелизма (второе обновление терпит неудачу) до более сложных подходов, таких как проверка индивидуальных полей. За исключением варианта бездействия (повсеместно считающегося признаком плохого стиля программирования) разработчики обязаны знать, когда возникают проблемы с параллелизмом, чтобы иметь возможность обработать их надлежащим образом.

К счастью, многие современные СУБД оснащены инструментами, которые помогают разработчикам решать проблемы с параллелизмом. В SQL Server имеется встроенный тип данных под названием `timestamp` — синоним для `rowversion`. Если столбец определен с типом данных `timestamp`, то при добавлении записи в базу данных значение для этого столбца создается СУБД SQL Server, а при обновлении записи значение столбца тоже обновляется. Фактически гарантируется, что значение будет уникальным и управляемым СУБД SQL Server.

В EF Core можно задействовать тип данных `timestamp` из SQL Server, реализуя внутри сущности свойство `Timestamp` (представляемое в C# как `byte[]`). Свойства сущностей, определенные с применением атрибута `Timestamp` либо Fluent API, предназначены для добавления в конструкцию `where` при обновлении или удалении запи-

сей. Вместо того чтобы просто использовать значение (значения) первичного ключа, в конструкцию `where` генерируемого оператора SQL добавляется значение свойства `timestamp`, что ограничивает результаты записями, у которых совпадают значения первичного ключа и отметки времени. Если запись была обновлена другим пользователем (или системой), тогда значения отметок времени не совпадут, так что оператор `update` не обновит, а оператор `delete` не удалит запись. Вот пример запроса обновления, в котором применяется столбец `Timestamp`:

```
UPDATE [Dbo].[Inventory] SET [Color] = N'Yellow'
WHERE [Id] = 1 AND [TimeStamp] = 0x0000000000000081F;
```

Когда хранилище сообщает о количестве затронутых записей, отличающемся от количества записей, изменения которых ожидает `ChangeTracker`, исполняющая среда EF Core генерирует исключение `DbUpdateConcurrencyException` и выполняет откат всей транзакции. Экземпляр `DbUpdateConcurrencyException` содержит информацию о записях, которые не были сохранены, куда входят первоначальные значения (полученные в результате загрузки из базы данных) и текущие значения (после их обновления пользователем/системой). Кроме того, существует метод для получения текущих значений в базе данных (требующий еще одного обращения к серверу). Располагая настолько большим количеством информации, разработчик затем может обработать ошибку параллелизма так, как того требует приложение. Ниже приведен пример:

```
try
{
    // Получить запись для автомобиля (неважно какую).
    var car = Context.Cars.First();
    // Обновить базу данных извне контекста.
    Context.Database.ExecuteSqlInterpolated($"Update dbo.Inventory set
Color='Pink' where Id = {car.Id}");
    // Обновить запись для автомобиля в ChangeTracker
    // и попробовать сохранить изменения.
    car.Color = "Yellow";
    Context.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
    // Получить сущность, которую не удалось обновить.
    var entry = ex.Entries[0];
    // Получить первоначальные значения (когда сущность была загружена).
    PropertyValues originalProps = entry.OriginalValues;
    // Получить текущие значения (обновленные кодом выше).
    PropertyValues currentProps = entry.CurrentValues;
    // Получить текущие значения из хранилища данных.
    // Примечание: это требует еще одного обращения к базе данных.
    // PropertyValues databaseProps = entry.GetDatabaseValues();
}
```

Устойчивость подключений

Кратковременные ошибки трудны в отладке и еще более трудны в воспроизведении. К счастью, многие поставщики баз данных имеют внутренний механизм повтора для сбоев в системе баз данных (проблемы с `tempdb`, ограничения пользователей и т.д.), который может быть задействован EF Core. Для SQL Server кратковременные

ошибки (согласно определению команды разработчиков СУБД) перехватываются экземпляром класса `SqlServerRetryingExecutionStrategy`, и если он включен в объекте производного от `DbContext` класса через `DbContextOptions`, то EF Core автоматически повторяет операцию до тех пор, пока не достигнет максимального предела повторов.

При работе с SQL Server доступен сокращенный метод, который можно использовать для включения `SqlServerRetryingExecutionStrategy` со всеми стандартными параметрами. Метод, который применяется с `SqlServerOptions` — это `EnableRetryOnFailure()`:

```
public ApplicationDbContext CreateDbContext(string[] args)
{
    var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
    var connectionString = @"server=.,5433;Database=AutoLot50;
User Id=sa;Password=P@ssw0rd;";
    optionsBuilder.UseSqlServer(connectionString,
        options => options.EnableRetryOnFailure());
    return new ApplicationDbContext(optionsBuilder.Options);
}
```

Максимальное количество повторов и предельное время между повторами можно конфигурировать в зависимости от требований приложения. Если предел повторов достигается без завершения операции, тогда EF Core уведомит приложение о проблемах с подключением путем генерации `RetryLimitExceededException`. В случае обработки это исключение способно передавать необходимую информацию пользователю, обеспечивая лучший отклик:

```
try
{
    Context.SaveChanges();
}
catch (RetryLimitExceededException ex)
{
    // Превышен предел повторов.
    // Требуется интеллектуальная обработка.
    Console.WriteLine($"Retry limit exceeded! {ex.Message}");
}
```

Для поставщиков баз данных, которые не предлагают встроенной стратегии выполнения, можно создавать специальную стратегию выполнения. Дополнительные сведения ищите в документации по EF Core: <https://docs.microsoft.com/ru-ru/ef/core/miscellaneous/connection-resiliency>.

Связанные данные

Навигационные свойства сущности используются для загрузки связанных данных сущности. Связанные данные можно загружать энергичным образом (один оператор LINQ, один запрос SQL), энергичным образом с разделением запросов (один оператор LINQ, множество запросов SQL), явным образом (множество вызовов LINQ, множество запросов SQL) или ленивым образом (один оператор LINQ, множество запросов SQL по требованию).

Помимо возможности загрузки связанных данных с применением навигационных свойств исполняющая среда EF Core будет автоматически приводить в порядок сущ-

ности по мере их загрузки в `ChangeTracker`. В качестве примера предположим, что все записи `Make` загружаются в `DbSet<Make>`, после чего все записи `Car` загружаются в `DbSet<Car>`. Несмотря на то что записи загружались по отдельности, они будут доступны друг другу через навигационные свойства.

Энергичная загрузка

Энергичная загрузка — это термин для обозначения загрузки связанных записей из множества таблиц в рамках одного обращения к базе данных. Прием аналогичен созданию запроса в T-SQL, связывающего две или большее число таблиц с помощью соединений. Когда сущности имеют навигационные свойства, которые используются в запросах LINQ, механизм трансляции применяет соединения, чтобы получить данные из связанных таблиц, и загружает соответствующие сущности. Такое решение обычно гораздо эффективнее, чем выполнение одного запроса с целью получения данных из одной таблицы и выполнение дополнительных запросов для каждой связанной таблицы. В ситуациях, когда использовать один запрос менее эффективно, в EF Core 5 предусмотрено разделение запросов, которое рассматривается далее.

Методы `Include()` и `ThenInclude()` (для последующих навигационных свойств) применяются для обхода навигационных свойств в запросах LINQ. Если отношение является обязательным, тогда механизм трансляции LINQ создаст внутреннее соединение. Если же отношение необязательное, то механизм трансляции создаст левое соединение.

Например, чтобы загрузить все записи `Car` со связанный информацией `Make`, запустите следующий запрос LINQ:

```
var queryable = Context.Cars.IgnoreQueryFilters().Include(
    c => c.MakeNavigation).ToList();
```

Предыдущий запрос LINQ выполняет в отношении базы данных такой запрос:

```
SELECT [i].[Id], [i].[Color], [i].[MakeId], [i].[PetName], [i].[TimeStamp],
    [m].[Id], [m].[Name], [m].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
INNER JOIN [dbo].[Makes] AS [m] ON [i].[MakeId] = [m].[Id]
```

В одном запросе можно использовать множество вызовов `Include()` для соединения исходной сущности сразу с несколькими сущностями. Чтобы спуститься вниз по дереву навигационных свойств, примените `ThenInclude()` после `Include()`. Скажем, для получения всех записей `Cars` со связанный информацией `Make` и `Order`, а также информацией `Customer`, связанной с `Order`, используйте показанный ниже оператор:

```
var cars = Context.Cars.Where(c => c.Orders.Any())
    .Include(c => c.MakeNavigation)
    .Include(c => c.Orders).ThenInclude(o => o.CustomerNavigation).ToList();
```

Фильтрованные включаемые данные

В версии EF Core 5 появилась возможность фильтрации и сортировки включаемых данных. Допустимыми операциями при навигации по коллекции являются `Where()`, `OrderBy()`, `OrderByDescending()`, `ThenBy()`, `ThenByDescending()`, `Skip()` и `Take()`. Например, если нужно получить все записи `Make`, но только со связанными записями `Car` с желтым цветом, тогда вы организуете фильтрацию навигационного свойства в лямбда-выражении такого вида:

```
var query = Context.Makes
    .Include(x => x.Cars.Where(x=>x.Color == "Yellow")).ToList();
```

В результате запустится следующий запрос:

```
SELECT [m].[Id], [m].[Name], [m].[TimeStamp], [t].[Id], [t].[Color],
[t].[MakeId], [t].[PetName], [t].[TimeStamp]
FROM [dbo].[Makes] AS [m]
LEFT JOIN (
    SELECT [i].[Id], [i].[Color], [i].[MakeId], [i].[PetName], [i].[TimeStamp]
    FROM [Dbo].[Inventory] AS [i]
    WHERE [i].[Color] = N'Yellow'
) AS [t] ON [m].[Id] = [t].[MakeId]
ORDER BY [m].[Id], [t].[Id]
```

Энергичная загрузка с разделением запросов

Наличие в запросе LINQ множества вызовов `Include()` может отрицательно повлиять на производительность. Для решения проблемы в EF Core 5 были введены разделяемые запросы. Вместо выполнения одиночного запроса исполняющая среда EF Core будет разделять запрос LINQ на несколько запросов SQL и затем объединять все связанные данные. Скажем, добавив к запросу LINQ вызов `AsSplitQuery()`, можно ожидать, что предыдущий запрос будет представлен в виде множества запросов SQL:

```
var query = Context.Makes.AsSplitQuery()
    .Include(x => x.Cars.Where(x=>x.Color == "Yellow")).ToList();
```

Вот как выглядят выполняемые запросы:

```
SELECT [m].[Id], [m].[Name], [m].[TimeStamp]
FROM [dbo].[Makes] AS [m]
ORDER BY [m].[Id]

SELECT [t].[Id], [t].[Color], [t].[MakeId], [t].[PetName],
[t].[TimeStamp], [m].[Id]
FROM [dbo].[Makes] AS [m]
INNER JOIN (
    SELECT [i].[Id], [i].[Color], [i].[MakeId], [i].[PetName],
    [i].[TimeStamp]
    FROM [Dbo].[Inventory] AS [i]
    WHERE [i].[Color] = N'Yellow'
) AS [t] ON [m].[Id] = [t].[MakeId]
ORDER BY [m].[Id]
```

Применению разделяемых запросов присущ и недостаток: если данные изменяются между выполнением запросов, тогда возвращаемые данные будут несогласованными.

Явная загрузка

Явная загрузка — это загрузка данных по навигационному свойству после того, как главный объект уже загружен. Такой процесс включает в себя дополнительное обращение к базе данных для получения связанных данных. Прием может быть удобен, если приложению необходимо получать связанные записи выборочно на основе какого-то действия пользователя, а не извлекать все связанные записи.

Процесс начинается с уже загруженной сущности и использования метода `Entry()` на экземпляре производного от `DbContext` класса. При запросе в отношении навига-

ционного свойства типа ссылки (например, с целью получения информации Make для автомобиля) применяйте метод `Reference()`. При запросе в отношении навигационного свойства типа коллекции используйте метод `Collection()`. Выполнение запроса откладывается до вызова `Load()`, `ToList()` или агрегирующей функции (вроде `Count()` либо `Max()`).

В следующих примерах показано, как получить связанные данные о производителе и заказах для записи `Car`:

```
// Получить запись Car.
var car = Context.Cars.First(x => x.Id == 1);

// Получить информацию о производителе.
Context.Entry(car).Reference(c => c.MakeNavigation).Load();

// Получить заказы, к которым относится данная запись Car.
Context.Entry(car).Collection(c => c.Orders).Query();
    IgnoreQueryFilters().Load();
```

Ленивая загрузка

Ленивая загрузка представляет собой загрузку записи по требованию, когда навигационное свойство применяется для доступа к связанной записи, которая пока еще не загружена в память. Ленивая загрузка — это средство EF 6, снова добавленное в версию EF Core 2.1. Хотя включение ленивой загрузки кажется разумной идеей, временами она может стать причиной возникновения проблем с производительностью в вашем приложении из-за потенциально лишних циклов взаимодействия с базой данных. В результате по умолчанию ленивая загрузка в EF Core отключена (в EF 6 она была включена).

Ленивая загрузка может быть полезна в приложениях интеллектуальных клиентов (WPF, Windows Forms), но в веб-приложениях и службах использовать ее не рекомендуется, так что в книге она не рассматривается. За дополнительными сведениями о ленивой загрузке и ее применением с EF Core обращайтесь в документацию по ссылке <https://docs.microsoft.com/ru-ru/ef/core/querying/related-data/lazy>.

Глобальные фильтры запросов

Глобальные фильтры запросов позволяют добавлять конструкцию `where` во все запросы LINQ для определенной сущности. Например, распространенное проектное решение для баз данных предусматривает использование “мягкого” удаления вместо “жесткого”. В таблицу добавляется поле, указывающее состояние удаления записи. Если запись “удалена”, то значение поля устанавливается в `true` (или `1`), но запись из базы данных не убирается. Прием называется “мягким” удалением. Чтобы отфильтровать записи, повергнувшиеся “мягкому” удалению, от тех, которые обрабатывались нормальными операциями, каждая конструкция `where` обязана проверять значение поля с состоянием удаления записи. Включение такого фильтра в каждый запрос может занять много времени, да и не забыть о нем довольно проблематично.

Инфраструктура EF Core позволяет добавлять к сущности глобальный фильтр запросов, который затем применяется к каждому запросу, вовлекающему эту сущность. Для описанного выше примера с “мягким” удалением вы устанавливаете фильтр на сущностном классе, чтобы исключить записи, повергнувшиеся “мягкому” удалению. К любым создаваемым EF Core запросам, затрагивающим сущности с глобальными фильтрами запросов, будут применяться их фильтры. Вам больше не придется помнить о необходимости включения конструкции `where` в каждый запрос.

Придерживаясь в книге темы автомобилей, предположим, что все записи Car, которые не являются управляемыми, должны отфильтровываться из нормальных запросов. Вот как можно добавить глобальный фильтр запросов с использованием Fluent API:

```
modelBuilder.Entity<Car>(entity =>
{
    entity.HasQueryFilter(c => c.IsDriveable == true);
    entity.Property(p => p.IsDriveable).HasField("_isDriveable").
        HasDefaultValue(true);
});
```

Благодаря такому глобальному фильтру запросы, вовлекающие сущность Car, будут автоматически отфильтровывать неуправляемые автомобили. Скажем, запуск следующего запроса LINQ:

```
var cars = Context.Cars.ToList();
```

приводит к выполнению показанного ниже оператора SQL:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp]
  FROM [Dbo].[Inventory] AS [i]
 WHERE [i].[IsDriveable] = CAST(1 AS bit)
```

Если вам нужно просмотреть отфильтрованные записи, тогда добавьте в запрос LINQ вызов IgnoreQueryFilters(), который отключает глобальные фильтры запросов для каждой сущности в запросе LINQ. Запуск представленного далее запроса LINQ:

```
var cars = Context.Cars.IgnoreQueryFilters().ToList();
```

инициирует выполнение следующего оператора SQL:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp]
  FROM [Dbo].[Inventory] AS [i]
```

Важно отметить, что вызов IgnoreQueryFilters() удаляет фильтр запросов для всех сущностей в запросе LINQ, в том числе и тех, которые задействованы в вызовах Include() или ThenInclude().

Глобальные фильтры запросов на навигационных свойствах

Глобальные фильтры запросов можно также устанавливать на навигационных свойствах. Пусть вам необходимо отфильтровать любые заказы, которые содержат экземпляр Car, представляющий неуправляемый автомобиль. Фильтр создается на навигационном свойстве CarNavigation сущности Order:

```
modelBuilder.Entity<Order>().HasQueryFilter(e =>
    e.CarNavigation.IsDriveable);
```

При выполнении стандартного запроса LINQ любые заказы, содержащие неуправляемый автомобиль, будут исключаться из результата. Ниже показан оператор LINQ и генерированный оператор SQL:

```
// Код C#
var orders = Context.Orders.ToList();
/* Генерированный запрос SQL */
SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp]
```

```
FROM [Dbo].[Orders] AS [o]
INNER JOIN (SELECT [i].[Id], [i].[IsDrivable]
    FROM [Dbo].[Inventory] AS [i]
    WHERE [i].[IsDrivable] = CAST(1 AS bit)) AS [t] ON [o].[CarId] = [t].[Id]
WHERE [t].[IsDrivable] = CAST(1 AS bit)
```

Для удаления фильтра запросов используйте вызов `IgnoreQueryFilters()`. Вот как выглядит модифицированный оператор LINQ и сгенерированный запрос SQL:

```
// Код C#
var orders = Context.Orders.IgnoreQueryFilters().ToList();
/* Сгенерированный запрос SQL */
SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp]
FROM [Dbo].[Orders] AS [o]
```

Здесь уместно предостеречь: исполняющая среда EF Core не обнаруживает циклические глобальные фильтры запросов, поэтому при добавлении фильтров запросов к навигационным свойствам соблюдайте осторожность.

Явная загрузка с глобальными фильтрами запросов

Глобальные фильтры запросов действуют и при явной загрузке связанных данных. Например, если вы хотите загрузить записи Car для Make, то фильтр `IsDrivable` предотвратит загрузку в память записей, представляющих неуправляемые автомобили. В качестве примера взгляните на следующий фрагмент кода:

```
var make = Context.Makes.First(x => x.Id == makeId);
Context.Entry(make).Collection(c=>c.Cars).Load();
```

К настоящему моменту не должен вызывать удивление тот факт, что сгенерированный оператор SQL включает фильтр для неуправляемых автомобилей:

```
SELECT [i].[Id], [i].[Color], [i].[IsDrivable],
    [i].[MakeId], [i].[PetName], [i].[TimeStamp]
FROM [Dbo].[Inventory] AS [i]
WHERE ([i].[IsDrivable] = CAST(1 AS bit)) AND ([i].[MakeId] = 1
```

С игнорированием фильтров запросов при явной загрузке данных связана небольшая загвоздка. Возвращаемым типом метода `Collection()` является `CollectionEntry<Make,Car>`, который явно не реализует интерфейс `IQueryable<T>`. Чтобы вызвать `IgnoreQueryFilters()`, сначала потребуется вызвать метод `Query()`, который возвращает экземпляр реализации `IQueryable<Car>`:

```
var make = Context.Makes.First(x => x.Id == makeId);
Context.Entry(make).Collection(c=>c.Cars).Query().IgnoreQueryFilters().Load();
```

Тот же процесс применяется в случае использования метода `Reference()` для извлечения данных из навигационного свойства типа коллекции.

Выполнение низкоуровневых запросов SQL с помощью LINQ

Иногда получить корректный оператор LINQ для компилируемого запроса сложнее, чем просто написать код SQL напрямую. К счастью, инфраструктура EF Core располагает механизмом, позволяющим выполнять низкоуровневые операторы SQL в `DbSet<T>`. Методы `FromSqlRaw()` и `FromSqlRawInterpolated()` принимают строку, которая становится основой запроса LINQ. Такой запрос выполняется на стороне сервера.

Если низкоуровневый оператор SQL не является завершающим (скажем, хранимой процедурой, пользовательской функцией или оператором, который использует общее табличное выражение или заканчивается точкой с запятой), тогда в запрос можно добавить дополнительные операторы LINQ. Дополнительные операторы LINQ наподобие конструкций `Include()`, `OrderBy()` или `Where()` будут объединены с первоначальным низкоуровневым обращением SQL и любыми глобальными фильтрами запросов, после чего весь запрос выполнится на стороне сервера.

При использовании одного из вариантов `FromSql*` () запрос должен формироваться с использованием схемы базы данных и имени таблицы, а не имен сущностей. Метод `FromSqlRaw()` отправит строку в том виде, в каком она записана. Метод `FromSqlInterpolated()` применяет интерполяцию строк C# и каждая интерполированная строка транслируется в параметр SQL. В случае использования переменных вы должны использовать версию с интерполяцией для обеспечения дополнительной защиты, присущей параметризованным запросам.

Предположим, что для сущности `Car` установлен глобальный фильтр запросов. Тогда показанный ниже оператор LINQ получит первую запись `Inventory` со значением `Id`, равным 1, включит связанные данные `Make` и отфильтрует записи, касающиеся неуправляемых автомобилей:

```
var car = Context.Cars
    .FromSqlInterpolated($"Select * from dbo.Inventory where Id = {carId}")
    .Include(x => x.MakeNavigation)
    .First();
```

Механизм трансляции LINQ to SQL объединяет низкоуровневый оператор SQL с остальными операторами LINQ и выполняют следующий запрос:

```
SELECT TOP(1) [c].[Id], [c].[Color], [c].[IsDriveable], [c].[MakeId],
    [c].[PetName], [c].[TimeStamp],
    [m].[Id], [m].[Name], [m].[TimeStamp]
FROM (Select * from dbo.Inventory where Id = 1) AS [c]
INNER JOIN [dbo].[Makes] AS [m] ON [c].[MakeId] = [m].[Id]
WHERE [c].[IsDriveable] = CAST(1 AS bit)
```

Имейте в виду, что есть несколько правил, которые необходимо соблюдать в случае применения низкоуровневых запросов SQL с LINQ.

- Запрос SQL должен возвращать данные для всех свойств сущностного типа.
- Имена столбцов должны совпадать с именами свойств, с которыми они сопоставляются (улучшение по сравнению с версией EF 6, где сопоставления игнорировались).
- Запрос SQL не может содержать связанные данные.

Пакетирование операторов

В EF Core значительно повышена производительность при сохранении изменений в базе данных за счет выполнения операторов в одном и более пакетов. В итоге объем взаимодействия между приложением и базой данных уменьшается, увеличивая производительность и потенциально сокращая затраты (скажем, для облачных баз данных, где за транзакции приходится платить).

Исполняющая среда EF Core пакетирует операторы создания, обновления и удаления с использованием табличных параметров. Количество операторов, которые пакетирует EF Core, зависит от поставщика баз данных. Например, в SQL Server пакети-

рование неэффективно для менее 4 и более 40 операторов. Независимо от количества пакетов все операторы по-прежнему выполняются в рамках транзакции. Размер пакета можно конфигурировать посредством `DbContextOptions`, но в большинстве ситуаций (если не во всех) рекомендуется позволять EF Core рассчитывать размер пакета самостоятельно.

Если бы вы вставляли четыре записи об автомобилях в одной транзакции, как показано ниже:

```
var cars = new List<Car>
{
    new Car { Color = "Yellow", MakeId = 1, PetName = "Herbie" },
    new Car { Color = "White", MakeId = 2, PetName = "Mach 5" },
    new Car { Color = "Pink", MakeId = 3, PetName = "Avon" },
    new Car { Color = "Blue", MakeId = 4, PetName = "Blueberry" },
};

Context.Cars.AddRange(cars);
Context.SaveChanges();
```

то исполняющая среда EF Core пакетировала бы операторы в одиночное обращение. Вот как выглядел бы генерированный запрос:

```
exec sp_executesql N'SET NOCOUNT ON;
DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
MERGE [Dbo].[Inventory] USING (
VALUES (@p0, @p1, @p2, 0),
(@p3, @p4, @p5, 1),
(@p6, @p7, @p8, 2),
(@p9, @p10, @p11, 3)) AS i ([Color], [MakeId], [PetName], _Position) ON 1=0
WHEN NOT MATCHED THEN
INSERT ([Color], [MakeId], [PetName])
VALUES (i.[Color], i.[MakeId], i.[PetName])
OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;

SELECT [t].[Id], [t].[IsDrivable], [t].[TimeStamp] FROM [Dbo].[Inventory] t
INNER JOIN @inserted0 i ON ([t].[Id] = [i].[Id])
ORDER BY [i].[_Position];
',N'@p0 nvarchar(50),@p1 int,@p2 nvarchar(50),@p3 nvarchar(50),
@p4 int,@p5 nvarchar(50),@p6 nvarchar(50),@p7 int,@p8 nvarchar(50),
@p9 nvarchar(50),@p10 int,@p11 nvarchar(50)',@p0=N'Yellow',@p1=1,
@p2=N'Herbie',@p3=N'White',@p4=2,@p5=N'Mach 5',@p6=N'Pink',@p7=3,
@p8=N'Avon',@p9=N'Blue',@p10=4,@p11=N'Blueberry'
```

Принадлежащие сущностные типы

Возможность применения класса C# в качестве свойства сущности с целью определения коллекции свойств для другой сущности впервые появилась в версии EF Core 2.0 и в последующих версиях постоянно обновлялась. Когда типы, помеченные атрибутом `[Owned]` или сконфигурированные посредством Fluent API, добавлены в виде свойств сущности, инфраструктура EF Core добавит все свойства из сущностного класса `[Owned]` к владеющему классу. В итоге увеличивается вероятность многократного использования кода C#.

“За кулисами” EF Core считает результат отношением “один к одному”. Принадлежащий класс является зависимой сущностью, а владеющий класс — главной сущностью. Хотя принадлежащий класс рассматривается как сущность, он не может существовать без владеющего класса. Имена столбцов из принадлежащего класса по умолчанию получают формат ИмяНавигационногоСвойства_имяСвойстваШаблонаСущности (например, PersonalNavigation_FirstName). Стандартные имена можно изменять с применением Fluent API.

Взгляните на приведенный далее класс Person (обратите внимание на атрибут [Owned]):

```
[Owned]
public class Person
{
    [Required, StringLength(50)]
    public string FirstName { get; set; } = "New";
    [Required, StringLength(50)]
    public string LastName { get; set; } = "Customer";
}
```

Он используется классом Customer:

```
[Table("Customers", Schema = "Dbo")]
public partial class Customer : BaseEntity
{
    public Person PersonalInformation { get; set; } = new Person();
    [JsonIgnore]
    [InverseProperty(nameof(CreditRisk.CustomerNavigation))]
    public IEnumerable<CreditRisk> CreditRisks { get; set; } =
        new List<CreditRisk>();
    [JsonIgnore]
    [InverseProperty(nameof(Order.CustomerNavigation))]
    public IEnumerable<Order> Orders { get; set; } = new List<Order>();
}
```

По умолчанию два свойства Person отображаются на столбцы с именами PersonalInformation_FirstName и PersonalInformation_LastName. Чтобы изменить это, добавьте в метод OnConfiguring() следующий код Fluent API:

```
modelBuilder.Entity<Customer>(entity =>
{
    entity.OwnsOne(o => o.PersonalInformation,
    pd =>
    {
        pd.Property<string>(nameof(Person.FirstName))
            .HasColumnName(nameof(Person.FirstName))
            .HasColumnType("nvarchar(50)");
        pd.Property<string>(nameof(Person.LastName))
            .HasColumnName(nameof(Person.LastName))
            .HasColumnType("nvarchar(50)");
    });
});
```

Вот как будет создаваться результирующая таблица (обратите внимание, что допустимость значений null в столбцах FirstName и LastName не соответствует аннотациям данных в принадлежащей сущности Person):

```

CREATE TABLE [dbo].[Customers](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](50) NULL,
    [LastName] [nvarchar](50) NULL,
    [TimeStamp] [timestamp] NULL,
    [FullName] AS (([LastName]+', ') + [FirstName]),
CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
    ALLOW_PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO

```

Проблема с принадлежащими сущностями, которая может быть не видна вам, но приводить к сложной ситуации, в версии EF Core 5 устранена. Легко заметить, что класс Person содержит аннотации данных Required для обоих своих свойств, но оба столбца SQL Server определены как допускающие NULL. Так происходит из-за проблемы с тем, каким образом система миграции транслирует принадлежащие сущности, когда они используются с необязательным отношением. Для исправления проблемы потребуется сделать отношение обязательным.

Решить задачу можно двумя способами. Первый — включить допустимость null на уровне проекта или в классах C#. Тогда навигационное свойство PersonalInformation становится не допускающим значение null, что исполняющая среда EF Core учитывает и соответствующим образом конфигурирует столбцы в принадлежащей сущности. Второй способ предусматривает добавление кода Fluent API для того, чтобы сделать навигационное свойство обязательным:

```

modelBuilder.Entity<Customer>(entity =>
{
    entity.OwnsOne(o => o.PersonalInformation,
        pd =>
    {
        pd.Property<string>(nameof(Person.FirstName))
            .HasColumnName(nameof(Person.FirstName))
            .HasColumnType("nvarchar(50)");
        pd.Property<string>(nameof(Person.LastName))
            .HasColumnName(nameof(Person.LastName))
            .HasColumnType("nvarchar(50)");
    });
    entity.Navigation(c => c.PersonalInformation).IsRequired(true);
});

```

Существуют дополнительные аспекты для исследования с помощью принадлежащих сущностей, в том числе коллекции, разбиение таблиц и вложение, но они выходят за рамки тематики настоящей книги. За более подробной информацией о принадлежащих сущностях обращайтесь в документацию по EF Core: <https://docs.microsoft.com/ru-ru/ef/core/modeling/owned-entities>.

Сопоставление с функциями базы данных

Функции SQL Server можно сопоставлять с методами C# и включать в операторы LINQ. В таком случае метод C# служит просто заполнителем, поскольку в генерируемый запрос SQL внедряется серверная функция. Поддержка сопоставления с табличными функциями была добавлена в EF Core к уже имеющейся поддержке сопоставления со скалярными функциями. Дополнительные сведения о сопоставлении с функциями базы данных ищите в документации: <https://docs.microsoft.com/ru-ru/ef/core/querying/user-defined-function-mapping>.

Команды CLI глобального инструмента EF Core

Глобальный инструмент командной строки EF Core под названием dotnet-ef содержит команды, необходимые для создания шаблонов существующих баз данных в коде, для создания/удаления миграций баз данных и для работы с базой данных (обновление, удаление и т.п.). Чтобы пользоваться глобальным инструментом dotnet-ef, вы должны его установить с помощью следующей команды (если вы прорабатывали материал главы с самого начала, то уже сделали это):

```
dotnet tool install --global dotnet-ef --version 5.0.1
```

На заметку! Из-за того, что EF Core 5 не является выпуском с долгосрочной поддержкой (LTS), при использовании глобальных инструментов EF Core 5 потребуется указывать версию.

Для проверки результата установки откройте окно командной строки и введите такую команду:

```
dotnet ef
```

Если глобальный инструмент был успешно установлен, тогда вы получите схематическое изображение единорога EF Core (талисмана команды разработчиков) и список доступных команд, подобный показанному ниже (на экране единорог выглядит лучше):

```
    _/\_
   / \ \
  | .   \| \
  | | | | )   \| \
  | | | | \_ / // \| \
  | | | | /   \\\ \| \
Entity Framework Core .NET Command-line Tools 5.0.1
Usage: dotnet ef [options] [command]
Options:
  --version      Show version information.
  -h|--help      Show help information.
  -v|--verbose   Show verbose output.
  --no-color     Don't colorize output.
  --prefix-output Prefix output with level.
Commands:
  database       Commands to manage the database.
  dbcontext      Commands to manage DbContext types.
  migrations     Commands to manage migrations.
Use "dotnet ef [command] --help" for more information about a command.
```

Использование: `dotnet ef [параметры] [команда]`

Параметры:

<code>--version</code>	Показать информацию о версии.
<code>-h --help</code>	Показать справочную информацию.
<code>-v --verbose</code>	Включить многословный вывод.
<code>--no-color</code>	Не использовать цвета в выводе.
<code>--prefix-output</code>	Снабжать вывод префиксами для выделения уровней.

Команды:

<code>database</code>	Команды для управления базой данных.
<code>dbcontext</code>	Команды для управления типами <code>DbContext</code> .
<code>migrations</code>	Команды для управления миграциями.

Для получения дополнительной информации о команде применяйте `dotnet ef [команда] --help`.

В табл. 22.9 описаны три основных команды глобального инструмента EF Core. С каждой основной командой связаны дополнительные подкоманды. Как и все команды .NET Core, каждая команда имеет развитую справочную систему, которая доступна после ввода `-h` вместе с командой.

Таблица 22.9. Основные команды глобального инструмента

Команда	Описание
<code>database</code>	Подкоманды этой команды предназначены для управления базой данных и включают <code>drop</code> и <code>update</code>
<code>dbcontext</code>	Подкоманды этой команды предназначены для управления типами <code>DbContext</code> и включают <code>scaffold</code> , <code>list</code> и <code>info</code>
<code>migrations</code>	Подкоманды этой команды предназначены для управления миграциями и включают <code>add</code> , <code>list</code> , <code>remove</code> и <code>script</code>

Команды EF Core выполняются в отношении файлов проектов .NET Core (не файлов решений). Целевой проект должен ссылаться на пакет NuGet инструментов EF Core по имени `Microsoft.EntityFrameworkCore.Design`. Команды работают с файлом проекта, расположенным в том же каталоге, где команды вводятся, или с файлом проекта из другого каталога в случае ссылки на него через параметры командной строки.

Для команд CLI глобального инструмента EF Core, которым требуется экземпляр класса, производного от `DbContext` (`Database` и `Migrations`), при наличии в проекте только одного такого экземпляра именно он и будет использоваться. Если экземпляров `DbContext` несколько, тогда конкретный экземпляр необходимо указывать в параметре командной строки. Экземпляр производного от `DbContext` класса будет создаваться с применением экземпляра класса, реализующего интерфейс `IDesignTimeDbContextFactory<TContext>`, если инструмент смог его обнаружить.

Если инструменту не удалось его найти, то экземпляр класса, производного от `DbContext`, будет создаваться с использованием конструктора без параметров. Если ни того и ни другого не существует, тогда команда потерпит неудачу. Обратите внимание, что вариант с конструктором без параметров требует наличия переопределенной версии `OnConfiguring()`, что не считается хорошей практикой.

Лучший (и на самом деле единственный) вариант — всегда создавать реализацию `IDesignTimeDbContextFactory<TContext>` для каждого класса, производного от `DbContext`, который присутствует в приложении.

В табл. 22.10 описаны общие параметры, доступные для команд EF Core. Многие команды принимают дополнительные параметры или аргументы.

Таблица 22.10. Параметры команд EF Core

Параметр (короткая форма длинная форма)	Описание
--c --context <класс, производный от DbContext>	Полностью заданное имя производного от DbContext класса, который должен использоваться. При наличии в проекте нескольких классов, производных от DbContext, этот параметр является обязательным
-p --project <проект>	Проект, подлежащий использованию (куда помещать файлы). По умолчанию принимается текущий рабочий каталог
-s --startup-project <проект>	Стартовый проект, подлежащий использованию (содержит производный от DbContext класс). По умолчанию принимается текущий рабочий каталог
-h --help	Отображает справочную информацию и все параметры
-v --verbose	Отображает многословный вывод

Чтобы вывести список всех аргументов и параметров для команды, введите dotnet ef <команда> -h в окне командной строки, например:

```
dotnet ef migrations add -h
```

На заметку! Важно отметить, что команды CLI — это не команды C#, а потому правила отмены символов обратной косой черты и кавычек здесь не применяются.

Команды для управления миграциями

Команды migrations используются для добавления, удаления, перечисления и создания сценариев миграций. После того, как миграция применена к базе данных, в таблице __EFMigrationsHistory создается запись. Команды для управления миграциями кратко описаны в табл. 22.11 и более подробно в последующих подразделах.

Таблица 22.11. Команды для управления миграциями EF Core

Команда	Описание
add	Создает новую миграцию на основе изменений относительно предыдущей миграции
remove	Проверяет, была ли применена последняя миграция в проекте к базе данных, и если нет, то удаляет файл миграции (и конструирующий ее файл), после чего выполняет откат класса моментального снимка до предыдущей миграции
list	Выводит список всех миграций для класса, производного от DbContext, вместе с их состоянием (применена или ожидает)
script	Создает сценарий SQL для всех, одной или диапазона миграций

Команда add

Команда `add` создает новую миграцию базы данных, основываясь на текущей объектной модели. Процесс исследует каждую сущность со свойством `DbSet<T>` в производном от `DbContext` классе (и каждую сущность, которая может быть достигнута из таких сущностей с использованием навигационных свойств) и выясняет, есть ли какие-то изменения, которые должны быть применены к базе данных. При наличии изменений генерируется надлежащий код для обновления базы данных. Вскоре вы узнаете об этом больше.

Команда `add` требует передачи параметра `name`, который используется при именовании созданного класса и файлов для миграции. В дополнение к общим параметрам параметр `-o <путь>` или `--output-dir <путь>` указывает, куда должны помещаться файлы миграции. Стандартный каталог называется `Migrations` и относителен к текущему пути.

Для каждой добавленной миграции создаются два файла с частичными определениями того же самого класса. Имена обоих файлов начинаются с отметки времени и наименования миграции, которое было указано в качестве параметра для команды `add`. Первый файл называется `<ГГГГММДДЧЧММСС>_<НаименованиеMиграции>.cs`, а второй — `<ГГГГММДДЧЧММСС>_<НаименованиеMиграции>.Designer.cs`. Отметка времени основана на том, когда файл был создан, и в точности совпадает для обоих файлов. Первый файл представляет код, сгенерированный для изменений базы данных в этой миграции, а конструирующий файл — код, который предназначен для создания и обновления базы данных на основе всех миграций до этой миграции включительно.

Главный файл содержит два метода, `Up()` и `Down()`. В методе `Up()` находится код для обновления базы данных с учетом изменений этой миграции. В методе `Down()` содержится код для выполнения отката изменений этой миграции. Ниже приведен неполный листинг начальной миграции, рассматриваемой ранее в главе (`One2Many`):

```
public partial class One2Many : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Make",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(type: "nvarchar(max)", nullable: true),
                TimeStamp = table.Column<byte[]>(type: "varbinary(max)",
                    nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Make", x => x.Id);
            });
        ...
        migrationBuilder.CreateIndex(
            name: "IX_Cars_MakeId",
            table: "Cars",
            column: "MakeId");
    }
}
```

```

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(name: "Cars");
    migrationBuilder.DropTable(name: "Make");
}
}

```

Как видите, метод Up() создает таблицы, столбцы, индексы и т.д. Метод Down() удаляет созданные элементы. По мере необходимости механизм миграции будет выдавать операторы alter, add и drop, чтобы гарантировать соответствие базы данных вашей модели.

Конструирующий файл содержит два атрибута, которые связывают частичные определения с именем файла и классом, производным от DbContext. Ниже показан фрагмент листинга конструирующего класса с упомянутыми атрибутами:

```

[DbContext(typeof(ApplicationDbContext))]
[Migration("20201230020509_One2Many")]
partial class One2Many
{
    protected override void BuildTargetModel(ModelBuilder modelBuilder)
    {
        ...
    }
}

```

Первая миграция создает внутри целевого каталога дополнительный файл, именуемый в соответствии с производным от DbContext классом, т.е. <ИмяПроизводногоОтDbContextКласса>ModelSnapshot.cs. Этот файл имеет формат, совпадающий с форматом конструирующего файла, и содержит код, который представляет собой итог всех миграций. При добавлении или удалении миграций данный файл автоматически обновляется, чтобы соответствовать изменениям.

На заметку! Крайне важно не удалять файлы миграций вручную. Удаление приведет к тому, что файл <ИмяПроизводногоОтDbContextКласса>ModelSnapshot.cs утратит синхронизацию с вашими миграциями, по существу нарушив их работу. Если вы собираетесь удалять файлы миграций вручную, тогда удалите их все и начните сначала. Для удаления миграции используйте команду remove, которая будет описана ниже.

Исключение таблиц из миграций

Если какая-то сущность задействована сразу в нескольких DbContext, то каждый DbContext будет создавать код в файлах миграций для любых изменений, вносимых в эту сущность. В результате возникает проблема, потому что второй сценарий миграции потерпит неудачу, если изменения уже внесены в базу данных. До выхода версии EF Core 5 единственным решением было ручное редактирование одного из файлов миграций с целью удаления таких изменений.

В версии EF Core 5 производный от DbContext класс может помечать сущность как исключенную из миграций, позволяя другому DbContext становиться системой записи для данной сущности. В следующем коде сущность исключается из миграций:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<LogEntry>().ToTable("Logs",
        t => t.ExcludeFromMigrations());
}

```

Команда `remove`

Команда `remove` применяется для удаления миграций из проекта и всегда работает с последней миграцией (основываясь на отметках времени миграций). При удалении миграции исполняющая среда EF Core проверяет, не была ли миграция применена к базе данных, с помощью таблицы `_EFMigrationsHistory`. Если миграция применялась, тогда процесс терпит неудачу. Если же миграция не применялась или была подвергнута откату, то она удаляется, а файл моментального снимка модели обновляется.

Команда `remove` не принимает какие-либо параметры (поскольку всегда работает с последней миграцией) и использует те же самые параметры, что и команда `add`, плюс дополнительный параметр `force (-f || --force)`, который обеспечивает выполнение отката последней миграции и ее удаление за один шаг.

Команда `list`

Команда `list` позволяет получить все миграции для класса, производного от `DbContext`. По умолчанию она выводит список всех миграций и запрашивает базу данных с целью выяснения, были ли они применены. Если миграции не применялись, то они будут помечены как ожидающие. Один из параметров команды `list` предназначен для передачи специальной строки подключения, а другой позволяет вообще не подключаться к базе данных и просто вывести список миграций (табл. 22.12).

Команда `script`

Команда `script` создает сценарий SQL на основе одной или большего количества миграций и принимает два необязательных параметра, которые указывают, с какой миграции начинать и на какой миграции заканчивать. Если ни один параметр не задан, то сценарий создается для всех миграций. Параметры описаны в табл. 22.13.

Таблица 22.12. Параметры команды `list`

Параметр	Описание
<code>--connection <подключение></code>	Строка подключения к базе данных. По умолчанию принимается та, что указана в экземпляре реализации <code>IDesignTimeDbContextFactory</code> или в методе <code>OnConfiguring ()</code> производного от <code>DbContext</code> класса
<code>--no-connect</code>	Заставляет пропустить обращение к базе данных на предмет применения миграций

Таблица 22.13. Параметры команды `script`

Параметр	Описание
<code><FROM></code>	Начальная миграция. По умолчанию принимается 0, т.е. первая миграция
<code><TO></code>	Конечная миграция. По умолчанию принимается последняя миграция

Если миграции не указаны, тогда созданный сценарий станет совокупным итогом всех миграций. В случае предоставления миграций сценарий будет содержать изменения между двумя миграциями (включительно). Каждая миграция помещается внутрь транзакции. Если в базе данных, где запускается команда `script`, табли-

ца `__EFMigrationsHistory` не существует, то она создается. Кроме того, она будет обновляться для соответствия выполненным миграциям. Вот несколько примеров:

```
// Создать сценарий для всех миграций.  
dotnet ef migrations script  
  
// Создать сценарий для миграций от начальной до Many2Many включительно  
dotnet ef migrations script 0 Many2Many
```

В табл. 22.14 представлены дополнительные параметры. Параметр `-o` позволяет указать файл для сценария (в каталоге, относительном к тому, где запускается команда), а параметр `-i` создает идемпотентный сценарий (который содержит проверку, применялась ли уже миграция, и если применялась, то пропускает ее). Параметр `--no-transactions` отключает добавление транзакций в сценарий.

Таблица 22.14. Дополнительные параметры команды `script`

Параметр (короткая форма длинная форма)	Описание
<code>-o --output <файл></code>	Задает файл, куда должен быть записан результирующий сценарий
<code>-i --idempotent</code>	Генерирует сценарий, в котором перед применением миграции осуществляется проверка, не применялась ли она ранее
<code>--no-transactions</code>	Отключает помещение каждой миграции внутрь транзакции

Команды для управления базой данных

Для управления базой данных предназначены две команды, `drop` и `update`. Команда `drop` удаляет базу данных, если она существует, а команда `update` обновляет базу данных с использованием миграций.

Команда `drop`

Команда `drop` удаляет базу данных, указанную в строке подключения внутри метода `OnConfiguring()` производного от `DbContext` класса. С помощью параметра `force` можно отключить запрос на подтверждение и принудительно закрыть все подключения (табл. 22.15).

Таблица 22.15. Параметры команды `drop`

Параметр (короткая форма длинная форма)	Описание
<code>-f --force</code>	Не выдавать запрос на подтверждение и принудительно закрывать все подключения
<code>--dry-run</code>	Показать, какая база данных будет удалена, но не удалять ее

Команда `update`

Команда `update` принимает параметр с именем миграции и обычные параметры. Она имеет один дополнительный параметр `--connection <подключение>`, поз-

воляющий использовать строку подключения, которая не была сконфигурирована заранее.

Если команда запускается без имени миграции, тогда она обновляет базу данных до самой последней миграции, при необходимости создавая саму базу. Если указано имя миграции, то база данных обновляется до этой миграции. Все предшествующие миграции, которые пока не применялись, также будут применены. Имена примененных миграций сохраняются в таблице `_EFMigrationsHistory`.

Если имя миграции имеет отметку времени, которая соответствует более раннему моменту, чем другие примененные миграции, тогда выполняется откат всех более поздних миграций. Когда в качестве имени миграции указывается 0, происходит откат всех миграций и база данных становится пустой (не считая таблицы `_EFMigrationsHistory`).

Команды для управления типами `DbContext`

Доступны четыре команды для управления типами `DbContext`. Три из них (`list`, `info`, `script`) работают с классами, производными от `DbContext`, в вашем проекте. Команда `scaffold` создает производный от `DbContext` класс и сущности из существующей базы данных. Все четыре команды описаны в табл. 22.16.

Для команд `list` и `info` доступны обычные параметры. Команда `list` выдает список классов, производных от `DbContext`, в целевом проекте. Команда `info` предоставляет детали об указанном производном от `DbContext` классе, в том числе строку подключения, имя поставщика и источник данных. Команда `script` генерирует сценарий SQL, который создает вашу базу данных на основе объектной модели, игнорируя любые имеющиеся миграции. Команда `scaffold` используется для анализа существующей базы данных и рассматривается в следующем разделе.

Таблица 22.16. Команды для управления типами `DbContext`

Команда	Описание
<code>info</code>	Получает информацию о типе <code>DbContext</code>
<code>list</code>	Выдает список доступных типов <code>DbContext</code>
<code>scaffold</code>	Создает тип <code>DbContext</code> и сущностные типы для базы данных
<code>script</code>	Генерирует сценарий SQL из <code>DbContext</code> на основе объектной модели, пропуская любые миграции

Команда `scaffold`

Команда `scaffold` создает из существующей базы данных классы C# (производные от `DbContext` и сущностные классы), дополненные аннотациями данных (если требуется) и командами Fluent API. В табл. 22.17 описаны два обязательных параметра: строка подключения к базе данных и полностью заданный поставщик (например, `Microsoft.EntityFrameworkCore.SqlServer`).

Таблица 22.17. Параметры команды `scaffold`

Параметр	Описание
<code>Connection</code>	Строка подключения к базе данных
<code>Provider</code>	Используемый поставщик баз данных EF Core (скажем, <code>Microsoft.EntityFrameworkCore.SqlServer</code>)

Кроме того, есть параметры, которые позволяют выбирать специфические схемы и таблицы, имя и пространство имен создаваемого класса, выходной каталог и пространство имен для генерируемых сущностных классов, а также многое другое. Предусмотрены и стандартные параметры. В табл. 22.18 перечислены расширенные параметры, которые далее обсуждаются более подробно.

Таблица 22.18. Параметры команды scaffold

Параметр (короткая форма длинная форма)	Описание
-d --data-annotations	Использовать для конфигурирования модели атрибуты (где это возможно). Если параметр опущен, тогда применяется только Fluent API
-c --context <имя>	Имя подлежащего созданию класса, производного от DbContext
--context-dir <путь>	Каталог, куда должен быть помещен файл класса, производного от DbContext, относительно каталога проекта. По умолчанию принимается имя базы данных
-f --force	Заменять файлы, существующие в целевом каталоге
-o --output-dir <путь>	Каталог, куда должны быть помещены генерируемые файлы сущностных классов, относительно каталога проекта
--schema <имя схемы>...	Схемы таблиц, для которых должны генерироваться сущностные классы
-t --table <имя таблицы>...	Таблицы, для которых должны генерироваться сущностные классы
--use-database-names	Использовать имена таблиц и столбцов прямо из базы данных
-n --namespaces <пространство имен>	Пространство имен для генерируемых сущностных классов. По умолчанию совпадает с именем каталога
--context-namespace <пространство имен>	Пространство имен для генерируемого класса, производного от DbContext. По умолчанию совпадает с именем каталога
--no-onconfiguring	Не генерировать метод OnConfiguring()
--no-pluralize	Не использовать средство перевода имен в множественное число

В версии EF Core 5.0 команда scaffold стала работать гораздо надежнее. Как видите, на выбор предлагается довольно много вариантов. Если выбран вариант с аннотациями данных (-d), тогда EF Core будет применять аннотации данных там, где это возможно, и заполнять отличия с использованием Fluent API. Если вариант с -d не выбран, то вся конфигурация (отличающаяся от соглашений) кодируется с помощью Fluent API. Вы можете указывать пространство имен, схему и местоположение для генерируемых файлов с сущностными классами и классом, производным от DbContext. Если вы не хотите создавать шаблон для целой базы данных, тогда можете выбрать определенные схемы и таблицы. Параметр --no-onconfiguring позволяет исключить метод OnConfiguring() из шаблонного класса, а параметр --no-pluralize отключает средство перевода имен в множественное число. Упомянутое средство пре-

вращает имена сущностей в единственном числе (`Car`) в имена таблиц во множественном числе (`Cars`) при создании миграций и имена таблиц во множественном числе в имена сущностей в единственном числе при создании шаблона.

Резюме

В настоящей главе вы начали ознакомление с инфраструктурой Entity Framework Core. В ней были исследованы аспекты, лежащие в основе EF Core, выполнения запросов и отслеживания изменений. Вы узнали о придании формы своей модели, соглашениях EF Core, аннотациях данных и Fluent API, а также о том, как их применение влияет на проектное решение для базы данных. Наконец, вы научились пользоваться мощным интерфейсом командной строки EF Core и глобальными инструментами.

Наряду с тем, что в этой главе было предложено много теоретических сведений и мало кода, следующая глава содержит главным образом код и совсем немного теории. По завершении проработки материалов главы 23 в вашем распоряжении появится законченный уровень доступа к данным AutoLot.

глава 23

Построение уровня доступа к данным с помощью Entity Framework Core

В предыдущей главе раскрывались детали и возможности инфраструктуры EF Core. Текущая глава сосредоточена на применении того, что вы узнали об инфраструктуре EF Core, для построения уровня доступа к данным AutoLot. В начале главы строятся шаблоны сущностей и производного от `DbContext` класса для базы данных из предыдущей главы. Затем используемый в проекте подход “сначала база данных” меняется на подход “сначала код”, а сущности обновляются до своей финальной версии и применяются к базе данных с использованием миграций EF Core. Последним изменением, внесенным в базу данных, будет воссоздание хранимой процедуры `GetPetName` и создание нового представления базы данных (в комплекте с соответствующей моделью представления), что делается с помощью миграций.

Следующий шаг — формирование хранилищ, обеспечивающих изолированный доступ для создания, чтения, обновления и удаления (`create, read, update, delete — CRUD`) базы данных. Далее в целях тестирования к проекту будет добавлен код инициализации данных вместе с выборочными данными. Остаток главы посвящен испытаниям уровня доступа к данным AutoLot посредством автоматизированных интеграционных тестов.

“Сначала код” или “сначала база данных”

Прежде чем приступить к построению уровня доступа к данным, давайте обсудим два способа работы с EF Core и базой данных: “сначала код” или “сначала база данных”. Оба они совершенно допустимы, а выбор применяемого подхода в значительной степени зависит от самой команды разработчиков.

Подход “сначала код” означает, что вы создаете и конфигурируете свои сущностные классы и производный от `DbContext` класс в коде и затем используете миграции для обновления базы данных. Подобным образом разрабатывается большинство новых проектов. Преимущество подхода “сначала код” заключается в том, что в ходе построения приложения сущности развиваются на основе имеющихся у него потреб-

ностей. Миграции поддерживают синхронизацию с базой данных, поэтому проектное решение базы данных эволюционирует вместе с приложением. Такой развивающийся процесс проектирования популярен в командах гибкой разработки, поскольку он обеспечивает создание нужных частей в надлежащее время.

Если у вас уже есть база данных или вы предпочитаете, чтобы проектное решение базы данных управляло разрабатываемым приложением, тогда должен применяться подход “сначала база данных”. Вместо создания производного от `DbContext` класса и всех сущностных классов вручную вы формируете шаблоны классов из базы данных. В случае изменения базы данных вам придется заново сформировать шаблоны классов для сохранения своего кода в синхронизированном с базой данных состоянии. Любой специальный код в сущностных классах или в классе, производном от `DbContext`, должен быть помещен в частичные классы, чтобы он не переписывался при повторном создании шаблонов классов. К счастью, именно по этой причине процесс формирования шаблонов строит частичные классы.

Какой бы подход вы ни выбрали, “сначала код” или “сначала база данных”, имейте в виду, что он является обязательством. Если вы используете подход “сначала код”, то все изменения вносятся в классы сущностей и контекста, а база данных обновляется с применением миграций. Если вы используете подход “сначала база данных”, то все изменения должны вноситься в базу данных, после чего будет требоваться повторное создание шаблонов классов. Приложив некоторые усилия по планированию, вы можете переключаться с подхода “сначала база данных” на подход “сначала код” (и наоборот), но не должны вручную вносить изменения в код и базу данных одновременно.

Создание проектов AutoLot.Dal и AutoLot.Models

Уровень доступа к данным AutoLot состоит из двух проектов, один из которых содержит код, специфичный для EF Core (производный от `DbContext` класс, фабрику контекстов, хранилища, миграции и т.д.), а другой — сущности и модели представлений. Создайте новое решение под названием `Chapter23_AllProjects` и добавьте в него проект библиотеки классов .NET Core по имени `AutoLot.Models`. Удалите стандартный класс, созданный шаблоном, и добавьте в проект следующие пакеты NuGet:

```
Microsoft.EntityFrameworkCore.Abstractions
System.Text.Json
```

Пакет `Microsoft.EntityFrameworkCore.Abstractions` обеспечивает доступ ко многим конструкциям EF Core (вроде аннотаций данных) и легковеснее пакета `Microsoft.EntityFrameworkCore`. Добавьте в решение еще один проект библиотеки классов .NET Core по имени `AutoLot.Dal`. Удалите стандартный класс, сгенерированный шаблоном, включите ссылку на проект `AutoLot.Models` и добавьте в проект перечисленные далее пакеты NuGet:

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Design
```

Пакет `Microsoft.EntityFrameworkCore` предоставляет общую функциональность для EF Core. Пакет `Microsoft.EntityFrameworkCore.SqlServer` предлагает поставщик данных SQL, а пакет `Microsoft.EntityFrameworkCore.Design` требуется для инструментов командной строки EF Core.

Чтобы выполнить все указанные ранее шаги в командной строке, введите показанные ниже команды (в каталоге, где хотите создать решение):

```
dotnet new sln -n Chapter23_AllProjects
dotnet new classlib -lang c# -n AutoLot.Models -o .\AutoLot.Models -f net5.0
dotnet sln .\Chapter23_AllProjects.sln add .\AutoLot.Models
dotnet add AutoLot.Models package Microsoft.EntityFrameworkCore.Abstractions
dotnet add AutoLot.Models package System.Text.Json
dotnet new classlib -lang c# -n AutoLot.Dal -o .\AutoLot.Dal -f net5.0
dotnet sln .\Chapter23_AllProjects.sln add .\AutoLot.Dal
dotnet add AutoLot.Dal reference AutoLot.Models
dotnet add AutoLot.Dal package Microsoft.EntityFrameworkCore
dotnet add AutoLot.Dal package Microsoft.EntityFrameworkCore.Design
dotnet add AutoLot.Dal package Microsoft.EntityFrameworkCore.SqlServer
dotnet add AutoLot.Dal package Microsoft.EntityFrameworkCore.Tools
```

На заметку! В случае работы на машине с операционной системой, отличающейся от Windows, используйте символ разделителя каталогов, который принят в вашей системе. Поступать так придется в отношении всех команд CLI, приводимых в настоящей главе.

После создания проектов обновите каждый файл *.csproj для включения ссылочных типов, допускающих null, из версии C# 8. Обновление выделено полужирным:

```
<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

Создание шаблонов для класса, производного от DbContext, и сущностных классов

Следующий шаг предусматривает формирование шаблонов для базы данных AutoLot из главы 21 с применением инструментов командной строки EF Core. Перейдите в каталог проекта AutoLot.Dal в окне командной строки или в консоли диспетчера пакетов Visual Studio.

На заметку! В папке Chapter_21 хранилища GitHub для этой книги находятся резервные копии базы данных, ориентированные на Windows и Docker. За инструкциями по восстановлению базы данных обращайтесь в главу 21.

Воспользуйтесь инструментами командной строки EF Core, чтобы сформировать для базы данных AutoLot шаблоны сущностных классов и класса, производного от DbContext. Вот как выглядит команда (которая должна вводиться в одной строке):

```
dotnet ef dbcontext scaffold "server=.,5433;Database=AutoLot;
User Id=sa;Password=P@ssw0rd;" 
Microsoft.EntityFrameworkCore.SqlServer
-d -c ApplicationDbContext --context-namespace
AutoLot.Dal.EfStructures --context-dir EfStructures
--no-onconfiguring -n AutoLot.Models.
Entities -o ..\AutoLot.Models\Entities
```

Предыдущая команда формирует шаблоны для базы данных, находящейся по заданной строке подключения (для контейнера Docker, применяемого в главе 21), с использованием поставщика баз данных SQL Server. Флаг `-d` заставляет, где возможно, отдавать предпочтение аннотациям данных (перед Fluent API). В `-c` указывается имя контекста, в `--context-namespaces` — пространство имен для контекста, в `--context-dir` — каталог (относительно каталога текущего проекта) для контекста. С помощью `--no-onconfiguring` исключается метод `OnConfiguring()`. В `-o` задается выходной каталог для файлов сущностных классов (относительно каталога текущего проекта) и, наконец, в `-n` — пространство имен для сущностных классов. Показанная выше команда помещает все сущности в каталог `Entities` проекта `AutoLot.Models`, а класс `ApplicationContext` — каталог `EfStructures` проекта `AutoLot.Dal`.

Вы заметите, что шаблон для хранимой процедуры не создавался. Если бы в базе данных присутствовали какие-то представления, то для них были бы созданы шаблоны сущностей без ключей. Поскольку в EF Core не предусмотрено конструкций, напрямую отображаемых на хранимые процедуры, создать шаблон невозможно. С применением EF Core можно создавать хранимые процедуры и другие объекты SQL, но на этот раз шаблоны формируются только для таблиц и представлений.

Переключение на подход “сначала код”

Теперь, имея базу данных, для которой сформированы сущности, самое время переключиться с подхода “сначала база данных” на подход “сначала код”. Для такого переключения должна быть создана фабрика контекстов, а также миграция из текущего состояния проекта. Затем либо база данных удаляется и воссоздается за счет применения миграции, либо действует фиктивная реализация для “обмана” инфраструктуры EF Core.

Создание фабрики экземпляров класса, производного от `DbContext`, на этапе проектирования

Как было указано в главе 22, инструменты командной строки EF Core используют реализацию `IDesignTimeDbContextFactory` для создания экземпляра класса, производного от `DbContext`. Создайте в каталоге `EfStructures` проекта `AutoLot.Dal` новый файл класса по имени `ApplicationContextFactory.cs`. Добавьте в файл класса следующие пространства имен:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
```

Детали фабрики раскрывались в предыдущей главе, а здесь представлен только код. Для информационных целей посредством дополнительного вызова `Console.WriteLine()` на консоль выводится строка подключения. Не забудьте привести строку подключения в соответствие со своей средой:

```
namespace AutoLot.Dal.EfStructures
{
    public class ApplicationContextFactory
        : IDesignTimeDbContextFactory<ApplicationContext>
    {
        public ApplicationContext CreateDbContext(string[] args)
```

```
{  
    var optionsBuilder =  
        new DbContextOptionsBuilder<ApplicationContext>();  
    var connectionString = @"server=.,5433;Database=AutoLot;  
User Id=sa;Password=P@ssw0rd;";  
    optionsBuilder.UseSqlServer(connectionString);  
    Console.WriteLine(connectionString);  
    return new ApplicationContext(optionsBuilder.Options);  
}  
}  
}
```

Создание начальной миграции

Вспомните, что первая миграция создаст три файла: два файла с частичным классом миграции и еще один с полным моментальным снимком модели. Введите в окне командной строки показанную далее команду, находясь в каталоге AutoLot.Dal, чтобы создать новую миграцию по имени Initial (используя экземпляр только что полученного класса ApplicationDbContext) и поместить файлы миграции в каталог EfStructures\Migrations проекта AutoLot.Dal:

```
dotnet ef migrations add Initial -o EfStructures\Migrations  
-c AutoLot.Dal.EfStructures.ApplicationDbContext
```

На заметку! Важно позаботиться о том, чтобы в сгенерированные файлы или базу данных не вносились изменения до тех пор, пока не будет создана и применена начальная миграция. Изменения на любой из сторон приведут к тому, что код и база данных утратят синхронизацию. После применения начальной миграции все изменения должны вноситься в базу данных через миграции EF Core.

Удостоверьтесь в том, что миграция была создана и ожидает применения, выполнив команду `list`:

```
dotnet ef migrations list -c AutoLot.Dal.EfStructures.ApplicationDbContext
```

Результат покажет, что миграция Initial ожидает обработки (ваша отметка времени будет другой). Стока подключения присутствует в выводе из-за вызова `Console.WriteLine()` в методе `CreateDbContext()`:

```
Build started...
Build succeeded.
server=.,5433;Database=AutoLot;User Id=sa;Password=P@ssw0rd;
20201231203939 Initial (Pending)
```

Применение миграции

Самый простой способ применения миграции к базе данных предусматривает ее удаление и повторное создание. Если вас он устраивает, тогда можете ввести приведенные ниже команды и перейти к чтению следующего раздела:

```
dotnet ef database drop -f  
dotnet ef database update Initial  
-c AutoLot.Dal.EfStructures.ApplicationDbContext
```

Если вариант с удалением и повторным созданием базы данных не подходит (скажем, в случае базы данных Azure SQL), то инфраструктуре EF Core необходимо обеспечить уверенность о том, что миграция была применена. К счастью, с помощью EF Core выполнить всю работу легко. Начните с создания из миграции сценария SQL, используя следующую команду:

```
dotnet ef migrations script --idempotent -o FirstMigration.sql
```

Важными частями сценария являются те, которые создают таблицу `__EFMigrationsHistory` и затем добавляют в нее запись о миграции, указывающую на ее применение. Скопируйте эти части в новый запрос внутри Azure Data Studio или SQL Server Manager Studio. Вот необходимый код SQL (отметка времени у вас будет отличаться):

```
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
    );
END;
GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20201231203939_Initial', N'5.0.1');
```

Если вы теперь запустите команду `list`, то она больше не будет отображать миграцию `Initial` как ожидающую обработки. После применения начальной миграции проект и база данных синхронизированы, а разработка будет продолжаться в стиле “сначала код”.

Обновление модели

В этом разделе все текущие сущности обновляются до своих финальных версий, к тому же добавляется сущность регистрации в журнале. Обратите внимание, что ваши проекты не смогут быть скомпилированы вплоть до завершения данного раздела.

Сущности

В каталоге `Entities` проекта `AutoLot.Models` вы обнаружите пять файлов, по одному для каждой таблицы в базе данных. Несложно заметить, что имена имеют форму единственного, а не множественного числа (как в базе данных). Это особенность версии EF Core 5, где средство перевода имен в множественное число по умолчанию включено при создании шаблонов сущностей для базы данных.

Изменения, которые вы внесете в сущностные классы, включают добавление базового класса, создание принадлежащего сущностного класса `Person`, исправление имен навигационных свойств и добавление ряда дополнительных свойств. Кроме того, вы добавите новую сущность для регистрации в журнале (которая будет использоваться в главах, посвященных ASP.NET Core). В предыдущей главе подробно рассматривались соглашения EF Core, аннотации данных и Fluent API, так что в текущей главе будут приводиться в основном листинги кода с краткими описаниями.

Класс *BaseEntity*

Класс *BaseEntity* будет содержать столбцы *Id* и *TimeStamp*, присутствующие в каждой сущности. Создайте новый каталог по имени *Base* в каталоге *Entities* проекта *AutoLot.Models*. Поместите в этот каталог новый файл *BaseEntity.cs* со следующим кодом:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AutoLot.Models.Entities.Base
{
    public abstract class BaseEntity
    {
        [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }

        [Timestamp]
        public byte[]? TimeStamp { get; set; }
    }
}
```

Все сущности, шаблоны которых созданы из базы данных *AutoLot*, будут обновлены для применения базового класса *BaseEntity*.

Принадлежащий сущностный класс *Person*

Сущности *Customer* и *CreditRisk* имеют свойства *FirstName* и *LastName*. Если в каждой сущности присутствуют одни и те же свойства, тогда можно извлечь выгоду от перемещения этих свойств в принадлежащие классы. Пример с двумя свойствами тривиален, но принадлежащие сущностные классы помогают сократить дублирование кода и повысить согласованность. В дополнение к двум свойствам внутри классов определяется еще одно свойство, отображаемое на вычисляемый столбец SQL Server.

Создайте в каталоге *Entities* проекта *AutoLot.Models* новый каталог по имени *Owned* и добавьте в него новый файл *Person.cs*, содержимое которого показано ниже:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace AutoLot.Models.Entities.Owned
{
    [Owned]
    public class Person
    {
        [Required, StringLength(50)]
        public string FirstName { get; set; } = "New";

        [Required, StringLength(50)]
        public string LastName { get; set; } = "Customer";

        [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
        public string? FullName { get; set; }
    }
}
```

Свойство FullName допускает null, т.к. до сохранения в базе данных новые сущности не будут иметь установленных значений. Финальная конфигурация свойства Fullname обеспечивается с использованием Fluent API.

Сущность Car (*Inventory*)

Для таблицы Inventory был создан шаблон сущностного класса по имени Inventory, но имя Car предпочтительнее. Исправить ситуацию легко: измените имя файла на Car.cs и имя класса на Car. Атрибут [Table] применяется корректно, так что нужно просто добавить схему dbo. Обратите внимание, что параметр Schema необязателен, поскольку по умолчанию для SQL Server принимается dbo, но он был включен ради полноты:

```
[Table("Inventory", Schema = "dbo")]
[Index(nameof(MakeId), Name = "IX_Inventory_MakeId")]
public partial class Car : BaseEntity
{
    ...
}
```

Обновите операторы using следующим образом:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
```

Унаследуйте класс Car от BaseEntity, после чего удалите свойства Id и TimeStamp, конструктор и директиву #pragma nullable disable. Вот как выглядит код класса после таких изменений:

```
namespace AutoLot.Models.Entities
{
    [Table("Inventory", Schema = "dbo")]
    [Index(nameof(MakeId), Name = "IX_Inventory_MakeId")]
    public partial class Car : BaseEntity
    {
        public int MakeId { get; set; }
        [Required]
        [StringLength(50)]
        public string Color { get; set; }
        [Required]
        [StringLength(50)]
        public string PetName { get; set; }
        [ForeignKey(nameof(MakeId))]
        [InverseProperty("Inventories")]
        public virtual Make Make { get; set; }
        [InverseProperty(nameof(Order.Car))]
        public virtual ICollection<Order> Orders { get; set; }
    }
}
```

В коде все еще присутствуют проблемы, которые необходимо устранить. Свойства Color и PetName определены как не допускающие null, но их значения не устанавливаются в конструкторе или не инициализируются в определении свойств. Проблема решается с помощью инициализаторов свойств. Кроме того, добавьте к свойству PetName атрибут [DisplayName], чтобы сделать название свойства более удобным для восприятия человеком. Обновите свойства, как показано ниже (изменения выделены полужирным):

```
[Required]
[StringLength(50)]
public string Color { get; set; } = "Gold";

[Required]
[StringLength(50)]
[DisplayName("Pet Name")]
public string PetName { get; set; } = "My Precious";
```

На заметку! Атрибут [DisplayName] используется инфраструктурой ASP.NET Core и будет описан в части VIII.

Навигационное свойство Make потребуется переименовать в MakeNavigation и сделать допускающим null, а в обратном навигационном свойстве вместо “магической” строки должно применяться выражение nameof языка C#. Наконец, нужно удалить модификатор virtual. После всех модификаций свойство приобретает следующий вид:

```
[ForeignKey(nameof(MakeId))]
[InverseProperty(nameof(Make.Cars))]
public Make? MakeNavigation { get; set; }
```

На заметку! Модификатор virtual необходим для ленивой загрузки. Поскольку ленивая загрузка в примерах книги не используется, модификатор virtual будет удаляться из всех свойств внутри уровня доступа к данным.

Для навигационного свойства Orders требуется атрибут [JsonIgnore], чтобы предотвратить циклические ссылки JSON при сериализации объектной модели. В шаблонном коде обратное навигационное свойство задействует выражение nameof, но его понадобится обновить, т.к. имена всех навигационных свойств типа ссылок будут содержать суффикс Navigation. Последнее изменение связано с тем, что свойство должно иметь тип IEnumerable<Order>, а не ICollection<Order>, и инициализироваться с применением нового экземпляра List<Order>. Изменение не является обязательным, потому что ICollection<Order> тоже будет работать. Более низкоуровневый тип IEnumerable<T> предпочтительнее использовать с навигационными свойствами типа коллекций I Enumerable<T> (поскольку интерфейсы IQueryable<T> и ICollection<T> унаследованы от I Enumerable<T>). Модифицируйте код, как показано далее:

```
[JsonIgnore]
[InverseProperty(nameof(Order.CarNavigation))]
public I Enumerable<Order> Orders { get; set; } = new List<Order>();
```

Затем добавьте свойство NotMapped, которое будет отображать значение Make экземпляра Car, устранив необходимость в классе CarViewModel из главы 21. Если

связанная информация Make была извлечена из базы данных с записью Car, то значение MakeName отображается. Если связанные данные не были извлечены, тогда для свойства отображается строка Unknown (т.е. производитель не известен). Как вы должны помнить, свойства с атрибутом [NotMapped] не относятся к базе данных и существуют только внутри сущности. Добавьте следующий код:

```
[NotMapped]
public string MakeName => MakeNavigation?.Name ?? "Unknown";
```

Переопределите ToString() для отображения информации о транспортном средстве:

```
public override string ToString()
{
    // Поскольку столбец PetName может быть пустым,
    // определить стандартное имя **No Name**.
    return $"{PetName ?? "**No Name**"} is a {Color} {MakeNavigation?.Name}
        with ID {Id}.";
```

Добавьте к свойству MakeId атрибуты [Required] и [DisplayName]. Несмотря на то что инфраструктура EF Core считает свойство MakeId обязательным, т.к. оно не допускает значение null, механизму проверки достоверности ASP.NET Core нужен атрибут [Required]. Ниже приведен модифицированный код:

```
[Required]
[DisplayName("Make")]
public int MakeId { get; set; }
```

Финальное изменение заключается в добавлении свойства IsDriveable типа bool, не допускающего значения null, с поддерживающим полем, допускающим null, и отображаемым именем:

```
private bool? _isDriveable;
[DisplayName("Is Driveable")]
public bool IsDriveable
{
    get => _isDriveable ?? false;
    set => _isDriveable = value;
}
```

На этом обновление сущностного класса Car завершено.

Сущность Customer

Для таблицы Customers был создан шаблонный сущностный класс по имени Customer. Приведите операторы using к следующему виду:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
using AutoLot.Models.Entities.Owned;
```

Унаследуйте класс Customer от BaseEntity и удалите свойства Id и TimeStamp. Удалите конструктор и директиву #pragma nullable disable, после чего добавьте

атрибут [Table] со схемой. Удалите свойства FirstName и LastName, т.к. они будут заменены принадлежащим сущностным классом Person. Вот как выглядит код в настоящий момент:

```
namespace AutoLot.Models.Entities
{
    [Table("Customers", Schema = "dbo")]
    public partial class Customer : BaseEntity
    {
        [InverseProperty(nameof(CreditRisk.Customer))]
        public virtual ICollection<CreditRisk> CreditRisks { get; set; }
        [InverseProperty(nameof(Order.Customer))]
        public virtual ICollection<Order> Orders { get; set; }
    }
}
```

Подобно сущностному классу Car в коде по-прежнему присутствуют проблемы, которые необходимо устранить, к тому же понадобится добавить принадлежащий сущностный класс. К навигационным свойствам нужно добавить атрибут [JsonIgnore], атрибуты обратных навигационных свойств потребуется обновить с использованием суффикса Navigation, типы необходимо изменить на IEnumerable<T> с инициализацией, а модификатор virtual удалить. Ниже показан модифицированный код:

```
[JsonIgnore]
[InverseProperty(nameof(CreditRisk.CustomerNavigation))]
public IEnumerable<CreditRisk> CreditRisks { get; set; } =
    new List<CreditRisk>();

[JsonIgnore]
[InverseProperty(nameof(Order.CustomerNavigation))]
public IEnumerable<Order> Orders { get; set; } = new List<Order>();
```

Осталось лишь добавить свойство с типом принадлежащего сущностного класса. Отношение будет позже сконфигурировано посредством Fluent API.

```
public Person PersonalInformation { get; set; } = new Person();
```

Итак, обновление сущностного класса Customer окончено.

Сущность Make

Для таблицы Makes был создан шаблонный сущностный класс по имени Make. Операторы using должны иметь следующий вид:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
```

Унаследуйте класс Make от BaseEntity и удалите свойства Id и TimeStamp. Удалите конструктор и директиву #pragma nullable disable, а затем добавьте атрибут [Table] со схемой. Вот текущий код сущностного класса:

```

namespace AutoLot.Models.Entities
{
    [Table("Makes", Schema = "dbo")]
    public partial class Make : BaseEntity
    {
        [Required]
        [StringLength(50)]
        public string Name { get; set; }
        [InverseProperty(nameof(Inventory.Make))]
        public virtual ICollection<Inventory> Inventories { get; set; }
    }
}

```

В представленном далее коде демонстрируется инициализированное свойство Name, не допускающее null, и скорректированное навигационное свойство Cars (обратите внимание на изменение имени Inventory на Car в выражении nameof):

```

[Required]
[StringLength(50)]
public string Name { get; set; } = "Ford";

[JsonIgnore]
[InverseProperty(nameof(Car.MakeNavigation))]
public IEnumerable<Car> Cars { get; set; } = new List<Car>();

```

На этом сущностный класс Make завершен.

Сущность CreditRisk

Для таблицы CreditRisks был создан шаблонный сущностный класс по имени CreditRisk. Приведите операторы using к такому виду:

```

using System.ComponentModel.DataAnnotations.Schema;
using AutoLot.Models.Entities.Base;
using AutoLot.Models.Entities.Owned;

```

Унаследуйте класс CreditRisk от BaseEntity и удалите свойства Id и TimeStamp. Удалите конструктор и директиву #pragma nullable disable и добавьте атрибут [Table] со схемой. Удалите свойства FirstName и LastName, т.к. они будут заменены прилежащим сущностным классом Person. Ниже показан обновленный код сущностного класса:

```

namespace AutoLot.Models.Entities
{
    [Table("CreditRisks", Schema = "dbo")]
    public partial class CreditRisk : BaseEntity
    {
        public Person PersonalInformation { get; set; } = new Person();
        public int CustomerId { get; set; }

        [ForeignKey(nameof(CustomerId))]
        [InverseProperty("CreditRisks")]
        public virtual Customer Customer { get; set; }
    }
}

```

Исправьте навигационное свойство, для чего удалите модификатор `virtual`, используйте выражение `nameof` в атрибуте `[InverseProperty]` и добавьте к имени свойства суффикс `Navigation`:

```
[ForeignKey(nameof(CustomerId))]
[InverseProperty(nameof(Customer.CreditRisks))]
public Customer? CustomerNavigation { get; set; }
```

Финальное изменение заключается в добавлении свойства с типом принадлежащего сущностного класса. Отношение будет позже сконфигурировано посредством Fluent API.

```
public Person PersonalInformation { get; set; } = new Person();
```

Итак, сущностный класс `CreditRisk` закончен.

Сущность `Order`

Для таблицы `Orders` был создан шаблонный сущностный класс по имени `Order`. Модифицируйте операторы `using` следующим образом:

```
using System;
using System.ComponentModel.DataAnnotations.Schema;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
```

Унаследуйте класс `Order` от `BaseEntity` и удалите свойства `Id` и `TimeStamp`. Удалите конструктор и директиву `#pragma nullable disable`, а затем добавьте атрибут `[Table]` со схемой. Вот текущий код сущностного класса:

```
namespace AutoLot.Models.Entities
{
    [Table("Orders", Schema = "dbo")]
    [Index(nameof(CarId), Name = "IX_Orders_CarId")]
    [Index(nameof(CustomerId), nameof(CarId),
        Name = "IX_Orders_CustomerId_CarId", IsUnique = true)]
    public partial class Order : BaseEntity
    {
        public int CustomerId { get; set; }
        public int CarId { get; set; }
        [ForeignKey(nameof(CarId))]
        [InverseProperty(nameof(Inventory.Orders))]
        public virtual Inventory.Car { get; set; }
        [ForeignKey(nameof(CustomerId))]
        [InverseProperty("Orders")]
        public virtual Customer { get; set; }
    }
}
```

К именам навигационных свойств `Car` и `Customer` необходимо добавить суффикс `Navigation`. Навигационное свойство `Car` должно иметь тип `Car`, а не `Inventory`. В выражении `nameof` в обратном навигационном свойстве нужно применять `Car.Orders` вместо `Inventory.Orders`. В навигационном свойстве `Customer` должно использоваться выражение `nameof` для `InverseProperty`. Оба свойства должны быть сделаны допускающими значение `null`. Модификатор `virtual` понадобится удалить.

```
[ForeignKey(nameof(CarId))]
[InverseProperty(nameof(Car.Orders))]
public Car? CarNavigation { get; set; }

[ForeignKey(nameof(CustomerId))]
[InverseProperty(nameof(Customer.Orders))]
public Customer? CustomerNavigation { get; set; }
```

На этом сущностный класс Order завершен.

На заметку! В данный момент проект должен нормально компилироваться. Проект AutoLot.Dal не скомпилируется до тех пор, пока не будет обновлен класс ApplicationDbContext.

Сущность SeriLogEntry

База данных нуждается в дополнительной таблице для хранения журнальных записей. В проектах ASP.NET Core из части VIII будет применяться инфраструктура ведения журналов SeriLog, и один из вариантов предусматривает помещение журнальных записей в таблицу SQL Server. Хотя таблица будет использоваться через несколько глав, имеет смысл добавить ее сейчас.

Эта таблица не связана ни с одной из остальных таблиц и не задействует класс BaseEntity. Добавьте в каталог Entities новый файл класса по имени SeriLogEntry.cs и поместите в него следующий код:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Xml.Linq;

namespace AutoLot.Models.Entities
{
    [Table("SeriLogs", Schema = "Logging")]
    public class SeriLogEntry
    {
        [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public int Id { get; set; }
        public string? Message { get; set; }
        public string? MessageTemplate { get; set; }

        [MaxLength(128)]
        public string? Level { get; set; }
        [DataType(DataType.DateTime)]
        public DateTime? TimeStamp { get; set; }
        public string? Exception { get; set; }
        public string? Properties { get; set; }
        public string? LogEvent { get; set; }
        public string? SourceContext { get; set; }
        public string? RequestPath { get; set; }
        public string? ActionName { get; set; }
        public string? ApplicationName { get; set; }
        public string? MachineName { get; set; }
        public string? FilePath { get; set; }
        public string? MemberName { get; set; }
        public int? LineNumber { get; set; }
    }
}
```

```
[NotMapped]
public XElement? PropertiesXml
    => (Properties != null) ? XElement.Parse(Properties):null;
}
}
```

Итак, сущностный класс SeriLogEntry закончен.

На заметку! Свойство TimeStamp в сущностном классе SeriLogEntry отличается от свойства TimeStamp в классе BaseEntity. Имена совпадают, но в этой таблице оно хранит дату и время регистрации записи в журнале (что будет конфигурироваться как стандартная настройка SQL Server), а не rowversion в других сущностях.

Класс ApplicationDbContext

Пришло время обновить файл ApplicationDbContext.cs. Начните с приведения операторов using к такому виду:

```
using System;
using System.Collections;
using System.Collections.Generic;
using AutoLot.Models.Entities;
using AutoLot.Models.Entities.Owned;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using AutoLot.Dal.Exceptions;
```

Файл начинается с конструктора без параметров. Удалите его, т.к. он не нужен. Следующий конструктор принимает экземпляр DbContextOptions и пока подходит. Перехватчики событий для DbContext и ChangeTracker добавляются позже в главе.

Свойства DbSet<T> необходимо сделать допускающими null, имена скорректировать, а модификаторы virtual удалить. Теперь новую сущность для ведения журнала необходимо добавить. Перейдите к свойствам DbSet<T> и модифицируйте их следующим образом:

```
public DbSet<SeriLogEntry>? LogEntries { get; set; }
public DbSet<CreditRisk>? CreditRisks { get; set; }
public DbSet<Customer>? Customers { get; set; }
public DbSet<Make>? Makes { get; set; }
public DbSet<Car>? Cars { get; set; }
public DbSet<Order>? Orders { get; set; }
```

Обновление кода Fluent API

Код Fluent API находится в переопределенной версии метода OnModelCreating() и использует экземпляра класса ModelBuilder для обновления модели.

Сущность SeriLogEntry

Первое изменение, вносимое в метод OnModelCreating(), касается добавления кода Fluent API для конфигурирования сущности SeriLogEntry. Свойство Properties является XML-столбцом SQL Server, а свойство TimeStamp отображается в SQL Server на столбец datetime2 со стандартным значением, установленным в

результат функции `getdate()` из SQL Server. Добавьте в метод `OnModelCreating()` следующий код:

```
modelBuilder.Entity<SerialLogEntry>(entity =>
{
    entity.Property(e => e.Properties).HasColumnType("Xml");
    entity.Property(e => e.TimeStamp).HasDefaultValueSql("GetDate()");
});
```

Сущность `CreditRisk`

Далее понадобится модифицировать код сущности `CreditRisk`. Блок конфигурирования для столбца `TimeStamp` удаляется, т.к. столбец конфигурируется в `BaseEntity`. Код конфигурирования навигации должен быть скорректирован с учетом новых имен. Кроме того, выполняется утверждение о том, что навигационное свойство не равно `null`. Другое изменение связано с конфигурированием свойства типа принадлежащей сущности, чтобы сопоставить с именами столбцов для `FirstName` и `LastName`, и добавлением вычисляемого значения для свойства `FullName`. Ниже показан обновленный блок для сущности `CreditRisk` с изменениями, выделенными полужирным:

```
modelBuilder.Entity<CreditRisk>(entity =>
{
    entity.HasOne(d => d.CustomerNavigation)
        .WithMany(p => p!.CreditRisks)
        .HasForeignKey(d => d.CustomerId)
        .HasConstraintName("FK_CreditRisks_Customers");

    entity.OwesOne(o => o.PersonalInformation,
        pd =>
    {
        pd.Property<string>(nameof(Person.FirstName))
            .HasColumnName(nameof(Person.FirstName))
            .HasColumnType("nvarchar(50)");
        pd.Property<string>(nameof(Person.LastName))
            .HasColumnName(nameof(Person.LastName))
            .HasColumnType("nvarchar(50)");
        pd.Property(p => p.FullName)
            .HasColumnName(nameof(Person.FullName))
            .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
    });
});
```

Сущность `Customer`

Следующим обновляется блок для сущности `Customer`. Здесь удаляется код для `TimeStamp` и конфигурируются свойства принадлежащего сущностного класса:

```
modelBuilder.Entity<Customer>(entity =>
{
    entity.OwesOne(o => o.PersonalInformation,
        pd =>
    {
        pd.Property(p
            => p.FirstName).HasColumnName(nameof(Person.FirstName));
        pd.Property(p
            => p.LastName).HasColumnName(nameof(Person.LastName));
    });
});
```

```

    pd.Property(p => p.FullName)
        .HasColumnName(nameof(Person.FullName))
        .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
    });
});

```

Сущность Make

Для сущности Make необходимо модифицировать блок конфигурирования, удалив свойство `TimeStamp` и добавив код, который ограничивает удаление сущности, имеющей зависимые сущности:

```

modelBuilder.Entity<Make>(entity =>
{
    entity.HasMany(e => e.Cars)
        .WithOne(c => c.MakeNavigation!)
        .HasForeignKey(k => k.MakeId)
        .onDelete(DeleteBehavior.Restrict)
        .HasConstraintName("FK_Make_Inventory");
});

```

Сущность Order

Для сущности `Order` обновите имена навигационных свойств и добавьте утверждение, что обратные навигационные свойства не равны `null`. Вместо ограничения удалений отношение `Customer` с `Order` настраивается на каскадное удаление:

```

modelBuilder.Entity<Order>(entity =>
{
    entity.HasOne(d => d.CarNavigation)
        .WithMany(p => p!.Orders)
        .HasForeignKey(d => d.CarId)
        .onDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_Orders_Inventory");
    entity.HasOne(d => d.CustomerNavigation)
        .WithMany(p => p!.Orders)
        .HasForeignKey(d => d.CustomerId)
        .onDelete(DeleteBehavior.Cascade)
        .HasConstraintName("FK_Orders_Customers");
});

```

Настройте фильтр для свойства `CarNavigation` сущности `Order`, чтобы отфильтровывать неуправляемые автомобили. Обратите внимание, что этот код находится не в том же блоке, где был предыдущий код. Никаких формальных причин разносить код не существует; просто здесь демонстрируется альтернативный синтаксис для конфигурирования в отдельных блоках:

```

modelBuilder.Entity<Order>().HasQueryFilter(e
    => e.CarNavigation!.IsDrivable);

```

Сущность Car

Шаблонный класс содержит конфигурацию для класса `Inventory`, который понадобится изменить на `Car`. Свойство `TimeStamp` нужно удалить, а в конфигурации навигационных свойств изменить имена свойств на `MakeNavigation` и `Cars`. Сущность получает фильтр запросов для отображения по умолчанию только управляемых автомобилей и устанавливает стандартное значение свойства `IsDrivable` в `true`. Код должен иметь следующий вид:

```

modelBuilder.Entity<Car>(entity =>

```

```
{
    entity.HasQueryFilter(c => c.IsDriveable);
    entity.Property(p
        => p.IsDriveable).HasField("_isDriveable").HasDefaultValue(true);
    entity.HasOne(d => d.MakeNavigation)
        .WithMany(p => p.Cars)
        .HasForeignKey(d => d.MakeId)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_Make_Inventory");
});
}
```

Специальные исключения

Распространенный прием для обработки исключений предусматривает перехват системного исключения (и/или исключения EF Core, как в текущем примере), его регистрацию в журнале и генерацию специального исключения. Если специальное исключение перехватывается вышеуказанным методом, то разработчику известно, что исключение уже было зарегистрировано в журнале, и необходимо только отреагировать на него надлежащим образом в коде.

Создайте в проекте AutoLot.Dal новый каталог по имени Exceptions и поместите в него четыре новых файла классов, CustomException.cs, CustomConcurrencyException.cs, CustomDbUpdateException.cs и CustomRetryLimitExceededException.cs, содержимое которых показано ниже:

```
// CustomException.cs
using System;
namespace AutoLot.Dal.Exceptions
{
    public class CustomException : Exception
    {
        public CustomException() {}
        public CustomException(string message) : base(message) { }
        public CustomException(string message, Exception innerException)
            : base(message, innerException) { }
    }
}

// CustomConcurrencyException.cs
using Microsoft.EntityFrameworkCore;
namespace AutoLot.Dal.Exceptions
{
    public class CustomConcurrencyException : CustomException
    {
        public CustomConcurrencyException() { }
        public CustomConcurrencyException(string message) : base(message) { }
        public CustomConcurrencyException(
            string message, DbUpdateConcurrencyException innerException)
            : base(message, innerException) { }
    }
}

// CustomDbUpdateException.cs
using Microsoft.EntityFrameworkCore;
namespace AutoLot.Dal.Exceptions
```

```
{
    public class CustomDbUpdateException : CustomException
    {
        public CustomDbUpdateException() { }
        public CustomDbUpdateException(string message) : base(message) { }
        public CustomDbUpdateException(
            string message, DbUpdateException innerException)
            : base(message, innerException) { }
    }
}

// CustomRetryLimitExceeded.cs
using System;
using Microsoft.EntityFrameworkCore.Storage;
namespace AutoLot.Dal.Exceptions
{
    public class CustomRetryLimitExceeded : CustomException
    {
        public CustomRetryLimitExceeded() { }
        public CustomRetryLimitExceeded(string message)
            : base(message) { }
        public CustomRetryLimitExceeded(
            string message, RetryLimitExceeded innerException)
            : base(message, innerException) { }
    }
}
```

На заметку! Обработка специальных исключений была подробно раскрыта в главе 7.

Переопределение метода *SaveChanges()*

Как обсуждалось в предыдущей главе, метод *SaveChanges()* базового класса *DbContext* сохраняет результаты операций изменения, добавления и удаления в базе данных. Переопределение этого метода позволяет инкапсулировать обработку исключений в одном месте. Располагая специальными исключениями, добавьте оператор *using* для *AutoLot.Dal.Exceptions* в начало файла *ApplicationDbContext.cs*, после чего переопределите метод *SaveChanges()*:

```
public override int SaveChanges()
{
    try
    {
        return base.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Произошла ошибка параллелизма.
        // Подлежит регистрации в журнале и надлежащей обработке.
        throw new CustomConcurrencyException("A concurrency error happened.", ex);
        // Произошла ошибка параллелизма.
    }
    catch (RetryLimitExceeded ex)
    {
        // Превышен предел на количество повторных попыток DbResiliency.
```

```

    // Подлежит регистрации в журнале и надлежащей обработке.
    throw new CustomRetryLimitExceededException(
        "There is a problem with SQL Server.", ex);
        // Возникла проблема с SQL Server.
}
catch (DbUpdateException ex)
{
    // Подлежит регистрации в журнале и надлежащей обработке.
    throw new CustomDbUpdateException(
        "An error occurred updating the database.", ex);
        // Произошла ошибка при обновлении базы данных.
}
catch (Exception ex)
{
    // Подлежит регистрации в журнале и надлежащей обработке.
    throw new CustomException(
        "An error occurred updating the database.", ex);
        // Произошла ошибка при обновлении базы данных.
}
}
}

```

Обработка событий *DbContext* и *ChangeTracker*

Перейдите к конструктору класса `ApplicationDbContext` и добавьте три события `DbContext`, которые обсуждались в предыдущей главе:

```

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
    base.SavingChanges += (sender, args) =>
    {
        Console.WriteLine($"Saving changes for {{({(ApplicationDbContext)} sender)!.Database!.GetConnectionString()}}");
    };
    base.SavedChanges += (sender, args) =>
    {
        Console.WriteLine($"Saved {args!.EntitiesSavedCount} changes for
{{({(ApplicationDbContext)} sender)!.Database!.GetConnectionString()}}");
    };
    base.SaveChangesFailed += (sender, args) =>
    {
        Console.WriteLine($"An exception occurred! {args.Exception.Message}");
    };
}

```

Затем добавьте обработчики для событий `StateChanged` и `Tracked` класса `ChangeTracker`:

```

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
    ...
    ChangeTracker.Tracked += ChangeTracker_Tracked;
    ChangeTracker.StateChanged += ChangeTracker_StateChanged;
}

```

Аргументы события Tracked содержат ссылку на сущность, которая инициировала событие, и указывают, было оно получено из запроса (загруженного из базы данных) или добавлено программно. Добавьте в класс ApplicationContext следующий обработчик событий:

```
private void ChangeTracker_Tracked(object? sender, EntityTrackedEventArgs e)
{
    var source = (e.FromQuery) ? "Database" : "Code";
    if (e.Entry.Entity is Car c)
    {
        Console.WriteLine($"Car entry {c.PetName} was added from {source}");
    }
}
```

Событие StateChanged инициируется при изменении состояния сущности. Одно из применений этого события — аудит. Поместите в класс ApplicationContext приведенный ниже обработчик событий. Если свойство NewState сущности имеет значение Unchanged, тогда выполняется проверка свойства OldState для выяснения, сущность была добавлена или же модифицирована.

```
private void ChangeTracker_StateChanged(object? sender,
                                         EntityStateChangedEventArgs e)
{
    if (e.Entry.Entity is not Car c)
    {
        return;
    }
    var action = string.Empty;
    Console.WriteLine($"Car {c.PetName} was {e.OldState} before the
state changed to {e.NewState}");
    switch (e.NewState)
    {
        case EntityState.Unchanged:
            action = e.OldState switch
            {
                EntityState.Added => "Added",
                EntityState.Modified => "Edited",
                _ => action
            };
            Console.WriteLine($"The object was {action}");
            break;
    }
}
```

Создание миграции и обновление базы данных

На этой стадии оба проекта компилируются и все готово к созданию еще одной миграции для обновления базы данных. Введите в каталоге проекта AutoLot.Dal следующие команды (каждая команда должна вводиться в одной строке):

```
dotnet ef migrations add UpdatedEntities -o EfStructures\Migrations
-c AutoLot.Dal.EfStructures.ApplicationDbContext

dotnet ef database update UpdatedEntities
-c AutoLot.Dal.EfStructures.ApplicationDbContext
```

Добавление представления базы данных и хранимой процедуры

Осталось внести в базу данных два изменения: создать хранимую процедуру GetPetName, рассмотренную в главе 21, и добавить представление базы данных, которое объединяет таблицу Orders с деталями Customer, Car и Make.

Добавление класса MigrationHelpers

Хранимая процедура и представление будут создаваться с использованием миграции, которая требует написания кода вручную. Причина поступать так (вместо того, чтобы просто открыть Azure Data Studio и запустить код T-SQL) — желание поместить полное конфигурирование базы данных в один процесс. Когда все содержится в миграциях, единственный вызов dotnet ef database update гарантирует, что база данных является актуальной, включая конфигурацию EF Core и специальный код SQL.

Выполнение команды the dotnet ef migrations add при отсутствии изменений в модели все равно приводит к созданию файлов миграции, имеющих правильную отметку времени, с пустыми методами Up () и Down (). Введите показанную ниже команду для создания пустой миграции (но не применения миграций):

```
dotnet ef migrations add SQL -o EfStructures\Migrations
    -c AutoLot.Dal.EfStructures.ApplicationDbContext
```

Создайте в каталоге EfStructures проекта AutoLot.Dal новый файл по имени MigrationHelpers.cs. Добавьте оператор using для пространства имен Microsoft.EntityFrameworkCore.Migrations, сделайте класс открытым и статическим и поместите в него следующие методы, которые используют MigrationBuilder для запуска операторов SQL в отношении базы данных:

```
namespace AutoLot.Dal.EfStructures
{
    public static class MigrationHelpers
    {
        public static void CreateSproc(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.Sql($"@"
                exec ('N'
                    CREATE PROCEDURE [dbo].[GetPetName]
                        @carID int,
                        @petName nvarchar(50) output
                AS
                    SELECT @petName = PetName from dbo.Inventory
                    where Id = @carID
                ');
            }
        public static void DropSproc(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.Sql("DROP PROCEDURE [dbo].[GetPetName]");
        }
        public static void CreateCustomerOrderView(
            MigrationBuilder migrationBuilder)
        {
```

```
migrationBuilder.Sql($@"
    exec (N'
CREATE VIEW [dbo].[CustomerOrderView]
AS
SELECT dbo.Customers.FirstName, dbo.Customers.LastName,
    dbo.Inventory.Color, dbo.Inventory.PetName,
    dbo.Inventory.IsDriveable,
    dbo.Makes.Name AS Make
FROM dbo.Orders
INNER JOIN dbo.Customers ON dbo.Orders.CustomerId = dbo.Customers.Id
INNER JOIN dbo.Inventory ON dbo.Orders.CarId = dbo.Inventory.Id
INNER JOIN dbo.Makes ON dbo.Makes.Id = dbo.Inventory.MakeId
')");
}

public static void DropCustomerOrderView(
    MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(
        "EXEC (N' DROP VIEW [dbo].[CustomerOrderView] ')");
}
```

Обновление и применение миграции

Для каждого объекта SQL Server в классе `MigrationHelpers` имеется два метода: один создает объект, другой удаляет объект. Вспомните, что при применении миграции выполняется метод `Up()`, а при откате миграции — метод `Down()`. Вызовы статических методов создания должны попасть в метод `Up()` миграции, тогда как вызовы статических методов удаления — в метод `Down()` миграции. В результате применения миграции создаются два объекта SQL Server, которые в случае отката миграции благополучно удаляются. Ниже приведен модифицированный код миграции:

```
namespace AutoLot.Dal.EfStructures.Migrations
{
    public partial class SQL : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            MigrationHelpers.CreateSproc(migrationBuilder);
            MigrationHelpers.CreateCustomerOrderView(migrationBuilder);
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            MigrationHelpers.DropSproc(migrationBuilder);
            MigrationHelpers.DropCustomerOrderView(migrationBuilder);
        }
    }
}
```

Если вы удалили свою базу данных, чтобы запустить начальную миграцию, тогда можете применить эту миграцию и двигаться дальше. Примените миграцию, выполнив следующую команду:

```
dotnet ef database update -c AutoIot.Dal.EfStructures.ApplicationDbContext
```

Если вы не удаляли свою базу данных для первой миграции, то процедура уже существует и не может быть снова создана. В таком случае легче всего закомментировать в методе Up() вызов статического метода, создающего хранимую процедуру:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    // MigrationHelpers.CreateSproc(migrationBuilder);
    MigrationHelpers.CreateCustomerOrderView(migrationBuilder);
}
```

После применения полученной миграции в первый раз уберите комментарий с указанной выше строки и все будет работать нормально. Разумеется, можно поступить и по-другому: удалить хранимую процедуру из базы данных и затем применить миграцию. В итоге нарушится парадигма "одно место для обновлений", но это часть перехода со способа "сначала база данных" на способ "сначала код".

На заметку! Вы также могли бы написать код, который сначала проверяет, существует ли объект, и в таком случае удаляет его, но это уже излишество для проблемы, которая возможна никогда не возникнет.

Добавление модели представления

Теперь, когда представление SQL Server на месте, самое время создать модель представления, которая будет использоваться для отображения данных из представления. Модель представления будет добавлена как DbSet<T> без ключа. Преимущество такого подхода в том, что данные можно запрашивать с помощью нормального процесса LINQ, общего для всех коллекций DbSet<T>.

Добавление класса модели представления

Добавьте в проект AutoLot.Models новый каталог по имени ViewModels, создайте в нем файл класса CustomerOrderViewModel.cs и поместите в него такие операторы using:

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;
```

Приведите код к следующему виду:

```
namespace AutoLot.Models.ViewModels
{
    [Keyless]
    public class CustomerOrderViewModel
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public string? Color { get; set; }
        public string? PetName { get; set; }
        public string? Make { get; set; }
        public bool? IsDrivable { get; set; }

        [NotMapped]
        public string FullDetail =>
            $"{FirstName} {LastName} ordered a {Color} {Make} named {PetName}";
        public override string ToString() => FullDetail;
    }
}
```

Аннотация данных [KeyLess] указывает, что класс является сущностью, работающей с данными, которые не имеют первичного ключа и могут быть оптимизированы как данные только для чтения (с точки зрения базы данных). Первые пять свойств соответствуют данным, поступающим из представления. Свойство FullDetail декорировано посредством аннотации данных [NotMapped], которая информирует инфраструктуру EF Core о том, что это свойство не должно включаться в базу данных, и не может поступать из базы данных в результате операций запросов. Инфраструктура EF Core также игнорирует переопределенную версию метода `ToString()`.

Добавление класса модели представления к `ApplicationContext`

Финальный шаг предусматривает регистрацию и конфигурирование `CustomerOrderViewModel` в `ApplicationContext`.

Добавьте к `ApplicationContext` оператор `using` для `AutoLot.Models.ViewModels` и затем свойство `DbSet<T>`:

```
public virtual DbSet<CustomerOrderViewModel>?
    CustomerOrderViewModels { get; set; }
```

Помимо добавления свойства `DbSet<T>` необходимо с помощью Fluent API сопоставить модель представления с представлением SQL Server. Метод `HasNoKey()` из Fluent API и аннотация данных [KeyLess] делают то же самое, но метод Fluent API замещает аннотацию данных. Ради ясности рекомендуется оставлять аннотацию данных на месте. Добавьте в метод `OnModelCreating()` следующий код:

```
modelBuilder.Entity<CustomerOrderViewModel>(entity =>
{
    entity.HasNoKey().ToView("CustomerOrderView", "dbo");
});
```

Добавление хранилищ

Распространенный паттерн проектирования для доступа к данным называется “Хранилище” (Repository). Согласно описанию Мартина Фаулера (<http://www.martinfowler.com/eaaCatalog/repository.html>) ядро этого паттерна является посредником между уровнями предметной области и сопоставления с данными. Наличие обобщенного хранилища, которое содержит общий код доступа к данным, помогает устраниТЬ дублирование кода. Наличие специфических хранилищ и интерфейсов, производных от базового хранилища, также хорошо подходит для работы с инфраструктурой внедрения зависимостей в ASP.NET Core.

Каждая сущность предметной области внутри уровня доступа к данным `AutoLot` будет иметь строго типизированное хранилище для инкапсуляции всей работы по доступу к данным. Первым делом создайте в проекте `AutoLot.Dal` новый каталог по имени `Repos`, предназначенный для хранения всех классов.

На заметку! Не воспринимайте следующий раздел как буквальную интерпретацию паттерна проектирования “Хранилище”. Если вас интересует исходный паттерн, который послужил мотивом для создания приведенной здесь версии, тогда почитайте о нем по ссылке <http://www.martinfowler.com/eaaCatalog/repository.html>.

Добавление базового интерфейса IRepo

Базовый интерфейс IRepo предоставляет множество общих методов, используемых при доступе к данным. Добавьте в проект AutoLot.Dal новый каталог по имени Repos и создайте в нем еще один каталог под названием Base. Поместите в каталог Repos\Base новый файл интерфейса по имени IRepo.cs. Обновите операторы using, как показано ниже:

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

Так выглядит полный интерфейс:

```
namespace AutoLot.Dal.Repos.Base
{
    public interface IRepo<T>: IDisposable
    {
        int Add(T entity, bool persist = true);
        int AddRange(IEnumerable<T> entities, bool persist = true);
        int Update(T entity, bool persist = true);
        int UpdateRange(IEnumerable<T> entities, bool persist = true);
        int Delete(int id, byte[] timeStamp, bool persist = true);
        int Delete(T entity, bool persist = true);
        int DeleteRange(IEnumerable<T> entities, bool persist = true);
        T? Find(int? id);
        T? FindAsNoTracking(int id);
        T? FindIgnoreQueryFilters(int id);
        IEnumerable<T> GetAll();
        IEnumerable<T> GetAllIgnoreQueryFilters();
        void ExecuteQuery(string sql, object[] sqlParametersObjects);
        int SaveChanges();
    }
}
```

Добавление класса BaseRepo

Добавьте в каталог Repos\Base файл класса по имени BaseRepo.cs. Класс BaseRepo будет реализовывать интерфейс IRepo и предлагать основную функциональность для хранилищ, специфичных к типам (рассматриваются далее). Приведите операторы using к следующему виду:

```
using System;
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Exceptions;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
```

Сделайте класс обобщенным с типом T и добавьте к нему ограничения BaseEntity и new(), что сузит набор типов до классов, которые имеют конструктор без параметров. Реализуйте интерфейс IRepo<T>:

```
public abstract class BaseRepo<T> : IRepo<T> where T : BaseEntity, new()
```

Классу хранилища нужен экземпляр ApplicationDbContext, внедренный через конструктор. В случае использования с контейнером внедрения зависимос-

тей ASP.NET Core временем жизни контекста будет управлять контейнер. Второй конструктор будет принимать `DbContextOptions` и должен создавать экземпляр `ApplicationDbContext`, который понадобится освобождать. Поскольку этот класс является абстрактным, оба конструктора определяются как защищенные. Добавьте в открытый класс `ApplicationContext` следующий код:

```
private readonly bool _disposeContext;
public ApplicationContext Context { get; }

protected BaseRepo(ApplicationContext context)
{
    Context = context;
    _disposeContext = false;
}

protected BaseRepo(DbContextOptions<ApplicationContext> options)
    : this(new ApplicationContext(options))
{
    _disposeContext = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

private bool _isDisposed;
protected virtual void Dispose(bool disposing)
{
    if (_isDisposed)
    {
        return;
    }
    if (disposing)
    {
        if (_disposeContext)
        {
            Context.Dispose();
        }
    }
    _isDisposed = true;
}

~BaseRepo()
{
    Dispose(false);
}
```

На свойства `DbSet<T>` класса `ApplicationContext` можно ссылаться с использованием метода `Context.Set<T>()`. Создайте открытое свойство по имени `Table` типа `DbSet<T>` и установите его начальное значение в конструкторе:

```
public DbSet<T> Table { get; }
protected BaseRepo(ApplicationContext context)
{
    Context = context;
    Table = Context.Set<T>();
    _disposeContext = false;
}
```

Реализация метода `SaveChanges()`

Класс `BaseRepo` имеет метод `SaveChanges()`, который вызывает переопределенную версию `SaveChanges()` и демонстрирует обработку специальных исключений. Добавьте в класс `BaseRepo` показанный ниже код:

```
public int SaveChanges()
{
    try
    {
        return Context.SaveChanges();
    }
    catch (CustomException ex)
    {
        // Подлежит надлежащей обработке -- уже зарегистрировано в журнале.
        throw;
    }
    catch (Exception ex)
    {
        // Подлежит регистрации в журнале и надлежащей обработке.
        throw new CustomException("An error occurred updating the database",
            ex);
    }
}
```

Реализация общих методов чтения

Следующий комплект методов возвращает записи с применением операторов LINQ. Метод `Find()` принимает первичный ключ (ключи) и сначала выполняет поиск в `ChangeTracker`. Если сущность уже отслеживается, тогда возвращается отслеживаемый экземпляр, иначе запись извлекается из базы данных.

```
public virtual T? Find(int? id) => Table.Find(id);
```

Дополнительные два метода `Find()` расширяют базовый метод `Find()`. Приведенный далее метод демонстрирует извлечение записи, но без ее добавления в `ChangeTracker`, используя `AsNoTrackingWithIdentityResolution()`. Добавьте в класс показанный ниже код:

```
public virtual T? FindAsNoTracking(int id) =>
    Table.AsNoTrackingWithIdentityResolution().FirstOrDefault(x => x.Id == id);
```

Другая вариация удаляет из сущности фильтры запросов и затем применяет сокращенную версию (пропускающую метод `Where()`) для получения `FirstOrDefault()`. Добавьте в класс следующий код:

```
public T? FindIgnoreQueryFilters(int id) =>
    Table.IgnoreQueryFilters().FirstOrDefault(x => x.Id == id);
```

Методы `GetAll()` возвращают все записи из таблицы. Первый метод извлекает их в порядке, поддерживаемом в базе данных, а второй по очереди обрабатывает все фильтры запросов:

```
public virtual IEnumerable<T> GetAll() => Table;
public virtual Ienumerable<T> GetAllIgnoreQueryFilters()
    => Table.IgnoreQueryFilters();
```

Метод `ExecuteQuery()` предназначен для выполнения хранимых процедур:

```
public void ExecuteQuery(string sql, object[] sqlParametersObjects)
    => Context.Database.ExecuteSqlRaw(sql, sqlParametersObjects);
```

Реализация методов добавления, обновления и удаления

Далее понадобится добавить блок кода, который будет служить оболочкой для соответствующих методов добавления, обновления и удаления, связанных со специфичным свойством `DbSet<T>`. Параметр `persist` определяет, выполняет ли хранилище вызов `SaveChanges()` сразу же после вызова методов добавления, обновления и удаления. Все методы помечены как `virtual`, чтобы сделать возможным дальнейшее переопределение. Добавьте в класс показанный ниже код:

```
public virtual int Add(T entity, bool persist = true)
{
    Table.Add(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int AddRange(IEnumerable<T> entities, bool persist = true)
{
    Table.AddRange(entities);
    return persist ? SaveChanges() : 0;
}
public virtual int Update(T entity, bool persist = true)
{
    Table.Update(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int UpdateRange(IEnumerable<T> entities,
                               bool persist = true)
{
    Table.UpdateRange(entities);
    return persist ? SaveChanges() : 0;
}
public virtual int Delete(T entity, bool persist = true)
{
    Table.Remove(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int DeleteRange(IEnumerable<T> entities, bool persist = true)
{
    Table.RemoveRange(entities);
    return persist ? SaveChanges() : 0;
}
```

Есть еще один метод удаления, который не следует этому шаблону. Для выдачи операции удаления он использует `EntityState`, что часто происходит при работе с ASP.NET Core с целью сокращения сетевого трафика:

```
public int Delete(int id, byte[] timeStamp, bool persist = true)
{
    var entity = new T {Id = id, TimeStamp = timeStamp};
    Context.Entry(entity).State = EntityState.Deleted;
    return persist ? SaveChanges() : 0;
}
```

Итак, класс BaseRepo завершен, и можно приступать к построению хранилищ, специфичных для сущностей.

Интерфейсы хранилищ, специфичных для сущностей

Каждая сущность будет иметь строго типизированное хранилище, производное от BaseRepo<T>, и интерфейс, который реализует IRepository<T>. Создайте в каталоге Repos проекта AutoLot.Dal новый каталог по имени Interfaces и добавьте в него пять файлов интерфейсов:

```
ICarRepo.cs
ICreditRiskRepo.cs
ICustomerRepo.cs
IMakeRepo.cs
IOrderRepo.cs
```

Содержимое интерфейсов будет представлено в последующих разделах.

Интерфейс хранилища данных об автомобилях

Откройте файл ICarRepo.cs и поместите в его начало такие операторы using:

```
using System.Collections.Generic;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
```

Измените интерфейс на public и реализуйте IRepository<Car>, как показано ниже:

```
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface ICarRepo : IRepository<Car>
    {
        IEnumerable<Car> GetAllBy(int makeId);
        string GetPetName(int id);
    }
}
```

Интерфейс хранилища данных о кредитных рисках

Откройте файл ICreditRiskRepo.cs. Интерфейс ICreditRiskRep не добавляет никакой функциональности сверх той, что предоставляется в BaseRepo. Обновите код следующим образом:

```
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface ICreditRiskRepo : IRepository<CreditRisk>
    {
    }
}
```

Интерфейс хранилища данных о заказчиках

Откройте файл ICustomerRepo.cs. Интерфейс ICustomerRepo не добавляет никакой функциональности сверх той, что предоставляется в BaseRepo. Приведите код к такому виду:

```
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface ICustomerRepo : IRepository<Customer>
    {
    }
}
```

Интерфейс хранилища данных о производителях

Откройте файл IMakeRepo.cs. Интерфейс IMakeRepo не добавляет никакой функциональности сверх той, что предоставляется в BaseRepo. Обновите код, как показано ниже:

```
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface IMakeRepo : IRepository<Make>
    {
    }
}
```

Интерфейс хранилища данных о заказах

Откройте файл IOrderRepo.cs. Поместите в начало файла следующие операторы using:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Models.ViewModels;
```

Измените интерфейс на public и реализуйте IRepository<Order>:

```
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface IOrderRepo : IRepository<Order>
    {
        IQueryable<CustomerOrderViewModel> GetOrdersViewModel();
    }
}
```

Интерфейс на этом завершен, т.к. все необходимые конечные точки API раскрыты в базовом классе.

Реализация классов хранилищ, специфичных для сущностей

Большую часть своей функциональности реализуемые классы хранилищ получают от базового класса. Далее будут описаны функциональные средства, которые добавляются или переопределяются возможностями, предлагаемые базовым классом хранилища. Создайте в каталоге Repos проекта AutoLot.Dal пять новых файлов классов хранилищ:

214 Часть VII. Entity Framework Core

```
CarRepo.cs  
CreditRiskRepo.cs  
CustomerRepo.cs  
MakeRepo.cs  
OrderRepo.cs
```

Классы хранилищ будут реализованы в последующих разделах.

Хранилище данных об автомобилях

Откройте файл класса CarRepo.cs и поместите в его начало показанные ниже операторы using:

```
using System.Collections.Generic;  
using System.Data;  
using System.Linq;  
using AutoLot.Dal.EfStructures;  
using AutoLot.Models.Entities;  
using AutoLot.Dal.Repos.Base;  
using AutoLot.Dal.Repos.Interfaces;  
using Microsoft.Data.SqlClient;  
using Microsoft.EntityFrameworkCore;
```

Измените класс на public, унаследуйте его от BaseRepo<Car> и реализуйте ICarRepo:

```
namespace AutoLot.Dal.Repos  
{  
    public class CarRepo : BaseRepo<Car>, ICarRepo  
    {  
    }  
}
```

Каждый класс хранилища должен реализовывать два конструктора из BaseRepo:

```
public CarRepo(DbContext context) : base(context)  
{  
}  
internal CarRepo(DbContextOptions<DbContext> options) :  
    base(options)  
{  
}
```

Добавьте переопределенные версии методов GetAll() и GetAllIgnoreQueryFilters() для включения свойства MakeNavigation и упорядочения по значению PetName:

```
public override IEnumerable<Car> GetAll()  
    => Table  
        .Include(c => c.MakeNavigation)  
        .OrderBy(o => o.PetName);  
  
public override IEnumerable<Car> GetAllIgnoreQueryFilters()  
    => Table  
        .Include(c => c.MakeNavigation)  
        .OrderBy(o => o.PetName)  
        .IgnoreQueryFilters();
```

Реализуйте метод GetAllBy(). Перед выполнением он обязан установить фильтр для контекста. Включите навигационное свойство Make и отсортируйте по значению PetName:

```
public IEnumerable<Car> GetAllBy(int makeId)
{
    return Table
        .Where(x => x.MakeId == makeId)
        .Include(c => c.MakeNavigation)
        .OrderBy(c => c.PetName);
}
```

Добавьте переопределенную версию Find(), в которой включается свойство MakeNavigation, а фильтры запросов игнорируются:

```
public override Car? Find(int? id)
=> Table
    .IgnoreQueryFilters()
    .Where(x => x.Id == id)
    .Include(m => m.MakeNavigation)
    .FirstOrDefault();
```

Добавьте метод, который позволяет получить значение PetName, используя хранимую процедуру:

```
public string GetPetName(int id)
{
    var parameterId = new SqlParameter
    {
        ParameterName = "@carId",
        SqlDbType = SqlDbType.Int,
        Value = id,
    };
    var parameterName = new SqlParameter
    {
        ParameterName = "@petName",
        SqlDbType = SqlDbType.NVarChar,
        Size = 50,
        Direction = ParameterDirection.Output
    };
    = Context.Database
        .ExecuteSqlRaw("EXEC [dbo].[GetPetName] @carId, @petName OUTPUT",
        parameterId,
        parameterName);
    return (string)parameterName.Value;
}
```

Хранилище данных о кредитных рисках

Откройте файл класса CreditRiskRepo.cs и поместите в его начало следующие операторы using:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

216 Часть VII. Entity Framework Core

Измените класс на public, унаследуйте его от BaseRepo<CreditRisk>, реализуйте ICreditRiskRepo и добавьте два обязательных конструктора:

```
namespace AutoLot.Dal.Repos
{
    public class CreditRiskRepo : BaseRepo<CreditRisk>, ICreditRiskRepo
    {
        public CreditRiskRepo(ApplicationDbContext context) : base(context)
        {
        }

        internal CreditRiskRepo(
            DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
    }
}
```

Хранилище данных о заказчиках

Откройте файл класса CustomerRepo.cs и поместите в его начало приведенные далее операторы using:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

Измените класс на public, унаследуйте его от BaseRepo<Customer>, реализуйте ICustomerRepo и добавьте два обязательных конструктора:

```
namespace AutoLot.Dal.Repos
{
    public class CustomerRepo : BaseRepo<Customer>, ICustomerRepo
    {
        public CustomerRepo(ApplicationDbContext context)
            : base(context)
        {
        }

        internal CustomerRepo(
            DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
    }
}
```

Наконец, добавьте метод, который возвращает все записи Customer с их заказами, отсортированные по значениям LastName:

```
public override IEnumerable<Customer> GetAll()
=> Table
    .Include(c => c.Orders)
    .OrderBy(o => o.PersonalInformation.LastName);
```

Хранилище данных о производителях

Откройте файл класса MakeRepo.cs и поместите в его начало перечисленные ниже операторы using:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

Измените класс на public, унаследуйте его от BaseRepo<Make>, реализуйте IMakeRepo и добавьте два обязательных конструктора:

```
namespace AutoLot.Dal.Repos
{
    public class MakeRepo : BaseRepo<Make>, IMakeRepo
    {
        public MakeRepo(ApplicationDbContext context)
            : base(context)
        {
        }

        internal MakeRepo(
            DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }
    }
}
```

Переопределите методы GetAll(), чтобы они сортировали значения Make по названиям:

```
public override IEnumerable<Make> GetAll()
    => Table.OrderBy(m => m.Name);
public override IEnumerable<Make> GetAllIgnoreQueryFilters()
    => Table.IgnoreQueryFilters().OrderBy(m => m.Name);
```

Хранилище данных о заказах

Откройте файл класса OrderRepo.cs и поместите в его начало следующие операторы using:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

Измените класс на public, унаследуйте его от BaseRepo<Order> и реализуйте IOrderRepo:

```
namespace AutoLot.Dal.Repos
{
    public class OrderRepo : BaseRepo<Order>, IOrderRepo
    {
```

```

public OrderRepo(ApplicationDbContext context)
    : base(context)
{
}
internal OrderRepo(
    DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
}
}
}
}

```

Реализуйте метод `GetOrderViewModel()`, который возвращает экземпляр реализации `IQueryable<CustomOrderViewModel>` из представления базы данных:

```

public IQueryable<CustomerOrderViewModel> GetOrdersViewModel()
{
    return Context.CustomerOrderViewModels!.AsQueryable();
}

```

На этом реализация всех классов хранилищ завершена. В следующем разделе будет написан код для удаления, создания и начального заполнения базы данных.

Программная работа с базой данных и миграциями

Свойство `Database` класса `DbContext` предлагает программные методы для удаления и создания базы данных, а также для запуска всех миграций. В табл. 23.1 описаны методы, соответствующие указанным операциям.

Как упоминалось в табл. 23.1, метод `EnsureCreated()` создает базу данных, если она не существует, после чего создает таблицы, столбцы и индексы на основе существенной модели. Никаких миграций он не применяет.

Таблица 23.1. Программная работа с базой данных

Метод <code>Database</code>	Описание
<code>EnsureDeleted()</code>	Удаляет базу данных, если она существует, иначе ничего не делает
<code>EnsureCreated()</code>	Создает базу данных, если она не существует, иначе ничего не делает. Таблицы и столбцы создаются на основе классов, достижимых через свойства <code>DbSet<T></code> . Не применяет никаких миграций. Примечание: должен использоваться в сочетании с миграциями
<code>Migrate()</code>	Создает базу данных, если она не существует. Применяет к базе данных все миграции

Если вы используете миграции, тогда при работе с базой данных будут возникать ошибки, и вам придется прибегнуть к уловке (как делалось ранее), чтобы заставить инфраструктуру EF Core “поверить” в то, что миграции были применены. Кроме того, вам нужно будет вручную применить к базе данных любые специальные объекты SQL. В случае работы с миграциями для программного создания базы данных всегда используйте метод `Migrate()`, а не `EnsureCreated()`.

Удаление, создание и очистка базы данных

Во время разработки нередко полезно удалять и воссоздавать рабочую базу данных и затем заполнять ее выборочными данными. В итоге получается среда, где тестирование (ручное или автоматизированное) может проводиться без опасения нарушить другие тесты из-за изменения данных. Создайте в проекте AutoLot.Dal новый каталог по имени Initialization и поместите в него новый файл класса SampleDataInitializer.cs. Вот как должны выглядеть операторы using в начале файла:

```
using System;
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
```

Сделайте класс открытым и статическим:

```
namespace AutoLot.Dal.Initialization
{
    public static class SampleDataInitializer
    {
    }
}
```

Создайте метод по имени DropAndCreateDatabase(), который в качестве единственного параметра принимает экземпляр ApplicationContext. Этот метод использует свойство Database экземпляра ApplicationContext, чтобы сначала удалить базу данных (с помощью метода EnsureDeleted()) и затем создать ее заново (посредством метода Migrate()):

```
public static void DropAndCreateDatabase(ApplicationDbContext context)
{
    context.Database.EnsureDeleted();
    context.Database.Migrate();
}
```

Создайте еще один метод по имени ClearData(), который удаляет все данные из базы данных и сбрасывает значения идентичности для первичного ключа каждой таблицы. Метод проходит по списку сущностей предметной области и применяет свойство Model класса DbContext для получения схемы и имени таблицы, на которые отображается каждая сущность. Затем он выполняет оператор DELETE и сбрасывает идентичность для каждой таблицы, используя метод ExecuteSqlRaw() на свойстве Database класса DbContext:

```
internal static void ClearData(ApplicationDbContext context)
{
    var entities = new[]
    {
        typeof(Order).FullName,
        typeof(Customer).FullName,
        typeof(Car).FullName,
        typeof(Make).FullName,
        typeof(CreditRisk).FullName
    };
```

```
foreach (var entityName in entities)
{
    var entity = context.Model.FindEntityType(entityName);
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
    context.Database.ExecuteSqlRaw($"DELETE FROM {schemaName}.{tableName}");
    context.Database.ExecuteSqlRaw($"DBCC CHECKIDENT ('{schemaName}.' + {tableName}', RESEED, 1);");
}
```

На заметку! Метод `ExecuteSqlRaw()` фасадного экземпляра базы данных должен применяться осторожно, чтобы избежать потенциальных атак внедрением в SQL.

Теперь, когда вы можете удалять и создавать базу данных и очищать данные, пора заняться методами, которые будут добавлять выборочные данные.

Инициализация базы данных

Вам предстоит построить свою систему заполнения начальными данными, которую можно запускать по требованию. Первым шагом будет создание выборочных данных и добавление в класс `SampleDataInitializer` методов для загрузки выборочных данных в базу.

Создание выборочных данных

Добавьте в каталог Initialization новый файл по имени SampleData.cs. Сделайте его открытым и статическим и поместите в него следующие операторы using:

```
using System.Collections.Generic;
using AutoLot.Dal.Entities;
using AutoLot.Dal.Entities.Owned;
namespace AutoLot.Dal.Initialization
{
    public static class SampleData
    {
    }
}
```

Класс `SampleData` содержит пять статических методов, которые создают выборочные данные:

```

public static List<Make> Makes => new()
{
    new() {Id = 1, Name = "VW"},  

    new() {Id = 2, Name = "Ford"},  

    new() {Id = 3, Name = "Saab"},  

    new() {Id = 4, Name = "Yugo"},  

    new() {Id = 5, Name = "BMW"},  

    new() {Id = 6, Name = "Pinto"},  

};

public static List<Car> Inventory => new()
{
    new() {Id = 1, MakeId = 1, Color = "Black", PetName = "Zippy"},  

    new() {Id = 2, MakeId = 2, Color = "Rust", PetName = "Rusty"},  

    new() {Id = 3, MakeId = 3, Color = "Black", PetName = "Mel"},  

    new() {Id = 4, MakeId = 4, Color = "Yellow", PetName = "Clunker"},  

    new() {Id = 5, MakeId = 5, Color = "Black", PetName = "Bimmer"},  

    new() {Id = 6, MakeId = 5, Color = "Green", PetName = "Hank"},  

    new() {Id = 7, MakeId = 5, Color = "Pink", PetName = "Pinky"},  

    new() {Id = 8, MakeId = 6, Color = "Black", PetName = "Pete"},  

    new() {Id = 9, MakeId = 4, Color = "Brown", PetName = "Brownie"},  

    new() {Id = 10, MakeId = 1, Color = "Rust", PetName = "Lemon",
           IsDrivable = false},
};

public static List<Order> Orders => new()
{
    new() {Id = 1, CustomerId = 1, CarId = 5},  

    new() {Id = 2, CustomerId = 2, CarId = 1},  

    new() {Id = 3, CustomerId = 3, CarId = 4},  

    new() {Id = 4, CustomerId = 4, CarId = 7},  

    new() {Id = 5, CustomerId = 5, CarId = 10},
};

public static List<CreditRisk> CreditRisks => new()
{
    new()
    {
        Id = 1,  

        CustomerId = Customers[4].Id,  

        PersonalInformation = new()  

        {
            FirstName = Customers[4].PersonalInformation.FirstName,  

            LastName = Customers[4].PersonalInformation.LastName
        }
    }
};

```

Загрузка выборочных данных

Внутренний метод `SeedData()` в классе `SampleDataInitializer` добавляет данные из методов класса `SampleData` к экземпляру `ApplicationDbContext` и сохраняет данные в базе данных:

```
internal static void SeedData(ApplicationDbContext context)
{
    try
    {
        ProcessInsert(context, context.Customers!, SampleData.Customers);
        ProcessInsert(context, context.Makes!, SampleData.Makes);
        ProcessInsert(context, context.Cars!, SampleData.Inventory);
        ProcessInsert(context, context.Orders!, SampleData.Orders);
        ProcessInsert(context, context.CreditRisks!, SampleData.CreditRisks);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        // Поместить сюда точку останова, чтобы выяснить,
        // в чем заключается проблема.
        throw;
    }
}

static void ProcessInsert<TEntity>(
    ApplicationDbContext context,
    DbSet<TEntity> table,
    List<TEntity> records) where TEntity : BaseEntity
{
    if (table.Any())
    {
        return;
    }
    IExecutionStrategy strategy = context.Database.CreateExecutionStrategy();
    strategy.Execute(() =>
    {
        using var transaction = context.Database.BeginTransaction();
        try
        {
            var metaData =
                context.Model.FindEntityType(typeof(TEntity).FullName);
            context.Database.ExecuteSqlRaw(
                $"SET IDENTITY_INSERT {metaData.GetSchema()}.{metaData.
GetTableName()} ON");
            table.AddRange(records);
            context.SaveChanges();
            context.Database.ExecuteSqlRaw(
                $"SET IDENTITY_INSERT {metaData.GetSchema()}.{metaData.
GetTableName()} OFF");
            transaction.Commit();
        }
        catch (Exception)
        {
            transaction.Rollback();
        }
    });
}
```

Для обработки данных в методе `SeedData()` используется локальная функция. Сначала она проверяет, содержит ли таблица какие-то записи, и если нет, то переходит к обработке выборочных данных. Из фасадного экземпляра базы данных создается экземпляр реализации `IExecutionStrategy`, применяемый для создания явной транзакции, которая необходима для включения и отключения вставки идентичности. Записи добавляются; если все прошло успешно, тогда транзакция фиксируется, а в противном случае подвергается откату.

Приведенные далее два открытых метода используются для сброса базы данных. Метод `InitializeData()` удаляет и воссоздает базу данных перед ее заполнением начальными данными, а метод `ClearDatabase()` просто удаляет все записи, сбрасывает идентичность и заполняет базу начальными данными:

```
public static void InitializeData(ApplicationDbContext context)
{
    DropAndCreateDatabase(context);
    SeedData(context);
}

public static void ClearAndReseedDatabase(ApplicationDbContext context)
{
    ClearData(context);
    SeedData(context);
}
```

Настройка тестов

Вместо создания клиентского приложения для испытания скомпилированного уровня доступа к данным `AutoLot` будет применяться автоматизированное интеграционное тестирование. Тесты продемонстрируют обращение к базе данных на предмет создания, чтения, обновления и удаления, что позволит исследовать код без накладных расходов по построению еще одного приложения. Каждый тест, рассматриваемый в этом разделе, будет выполнять запрос (создание, чтение, обновление или удаление) и иметь один и более операторов `Assert` для проверки, получен ли ожидаемый результат.

Создание проекта

Первым делом необходимо настроить платформу интеграционного тестирования с использованием `xUnit` — инфраструктуры тестирования, совместимой с .NET Core. Начните с добавления нового проекта тестирования `xUnit` по имени `AutoLot.Dal.Tests`, который в Visual Studio носит название `xUnit Test Project (.NET Core)` (Проект тестирования `xUnit (.NET Core)`).

На заметку! Модульные тесты предназначены для тестирования одной единицы кода.

Формально повсюду в главе создаются интеграционные тесты, т.к. производится тестирование кода C# и EF Core на всем пути к базе данных и обратно.

Введите следующую команду в окне командной строки:

```
dotnet new xunit -lang c# -n AutoLot.Dal.Tests -o .\AutoLot.Dal.Tests
-f net5.0 dotnet sln .\Chapter23_AllProjects.sln add AutoLot.Dal.Tests
```

224 Часть VII. Entity Framework Core

Добавьте в проект AutoLot.Dal.Tests перечисленные ниже пакеты NuGet:

```
Microsoft.EntityFrameworkCore  
Microsoft.EntityFrameworkCore.SqlServer  
Microsoft.Extensions.Configuration.Json
```

Поскольку версия пакета Microsoft.NET.Test.Sdk, поставляемая с шаблоном проектов xUnit, обычно отстает от текущей доступной версии, воспользуйтесь диспетчером пакетов NuGet для обновления всех пакетов NuGet. Затем добавьте ссылки на проекты AutoLot.Models и AutoLot.Dal.

В случае работы с CLI выполните приведенные далее команды (обратите внимание, что команды удаляют и повторно добавляют пакет Microsoft.NET.Test.Sdk, чтобы гарантировать ссылку на самую последнюю версию):

```
dotnet add AutoLot.Dal.Tests package Microsoft.EntityFrameworkCore  
dotnet add AutoLot.Dal.Tests  
    package Microsoft.EntityFrameworkCore.SqlServer  
dotnet add AutoLot.Dal.Tests  
    package Microsoft.Extensions.Configuration.Json  
dotnet remove AutoLot.Dal.Tests package Microsoft.NET.Test.Sdk  
dotnet add AutoLot.Dal.Tests package Microsoft.NET.Test.Sdk  
dotnet add AutoLot.Dal.Tests reference AutoLot.Dal  
dotnet add AutoLot.Dal.Tests reference AutoLot.Models
```

Конфигурирование проекта

Для извлечения строки подключения во время выполнения будут задействованы конфигурационные возможности .NET Core, предусматривающие работу с файлом JSON. Добавьте в проект файл JSON по имени appsettings.json и поместите в него информацию о своей строке подключения в следующем формате (надлежащим образом скорректировав ее):

```
{  
    "ConnectionStrings": {  
        "AutoLot": "server=.,5433;Database=AutoLotFinal;  
                    User Id=sa;Password=P@ssw0rd;"  
    }  
}
```

Модифицируйте файл проекта, чтобы файл appsettings.json копировался в выходной каталог при каждой компиляции проекта, для чего добавьте в файл AutoLot.Dal.Tests.csproj такой элемент ItemGroup:

```
<ItemGroup>  
    <None Update="appsettings.json">  
        <CopyToOutputDirectory>Always</CopyToOutputDirectory>  
    </None>  
</ItemGroup>
```

Создание класса TestHelpers

Класс TestHelpers будет обрабатывать конфигурацию приложения, а также создавать новый экземпляр ApplicationDbContext. Создайте в корневом каталоге проекта новый файл открытого статического класса по имени TestHelpers.cs. Приведите операторы using к следующему виду:

```
using System.IO;
using AutoLot.Dal.EfStructures;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Configuration;

namespace AutoLot.Dal.Tests
{
    public static class TestHelpers
    {
    }
}
```

Определите два открытых статических метода, предназначенные для создания экземпляров реализации `IConfiguration` и класса `ApplicationContext`. Добавьте в класс показанный ниже код:

```
public static IConfiguration GetConfiguration() =>
    new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", true, true)
        .Build();

public static ApplicationContext GetContext(
    IConfiguration configuration)
{
    var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();
    var connectionString = configuration.GetConnectionString("AutoLot");
    optionsBuilder.UseSqlServer(connectionString,
        sqlOptions => sqlOptions.EnableRetryOnFailure());
    return new ApplicationContext(optionsBuilder.Options);
}
```

Как вероятно вы помните, выделенный полужирным вызов `EnableRetryOnFailure()` выбирает стратегию повтора SQL Server, которая будет автоматически повторять операции, потерпевших неудачу из-за кратковременных ошибок.

Добавьте еще один статический метод, который будет создавать новый экземпляр `ApplicationContext` с применением того же самого подключения и транзакции, что и в переданном исходном контексте. Этот метод демонстрирует способ создания экземпляра `ApplicationContext` из существующего экземпляра с целью совместного использования подключения и транзакции:

```
public static ApplicationContext GetSecondContext(
    ApplicationContext oldContext,
    IDbContextTransaction trans)
{
    var optionsBuilder = new DbContextOptionsBuilder<ApplicationContext>();
    optionsBuilder.UseSqlServer(
        oldContext.Database.GetDbConnection(),
        sqlServerOptions => sqlServerOptions.EnableRetryOnFailure());
    var context = new ApplicationContext(optionsBuilder.Options);
    context.Database.UseTransaction(trans.GetDbTransaction());
    return context;
}
```

Добавление класса `BaseTest`

Создайте в проекте новый каталог по имени `Base` и добавьте туда новый файл класса `BaseTest.cs`. Модифицируйте операторы `using` следующим образом:

```
using System;
using System.Data;
using AutoLot.Dal.EfStructures;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Configuration;
```

Сделайте класс абстрактным и реализующим `IDisposable`. Добавьте два защищенных свойства `readonly` для хранения экземпляров реализации `IConfiguration` и класса `ApplicationContext` и освободите экземпляр `ApplicationContext` в виртуальном методе `Dispose()`:

```
namespace AutoLot.Dal.Tests.Base
{
    public abstract class BaseTest : IDisposable
    {
        protected readonly IConfiguration Configuration;
        protected readonly ApplicationContext Context;
        public virtual void Dispose()
        {
            Context.Dispose();
        }
    }
}
```

Инфраструктура тестирования xUnit предоставляет механизм для запуска кода до и после прогона *каждого* теста. Классы тестов (называемые *оснастками*), которые реализуют интерфейс `IDisposable`, перед прогоном каждого теста будут выполнять код в конструкторе класса (в конструкторе базового класса и конструкторе производного класса в этом случае), называемый *настройкой теста*, а после прогона каждого теста — код в методе `Dispose()` (в производном и в базовом классах), называемый *освобождением теста*.

Добавьте защищенный конструктор, который создает экземпляр реализации `IConfiguration` и присваивает его защищенной переменной класса. С применением конфигурации создайте экземпляр `ApplicationContext`, используя класс `TestHelpers`, и присвойте его защищенной переменной класса:

```
protected BaseTest()
{
    Configuration = TestHelpers.GetConfiguration();
    Context = TestHelpers.GetContext(Configuration);
}
```

Добавление вспомогательных методов для выполнения тестов в транзакциях

Последние два метода в классе `BaseTest` позволяют выполнять тестовые методы в транзакциях. Методы будут принимать в единственном параметре делегат `Action`, создавать явную транзакцию (или вовлекать существующую транзакцию), выполнять делегат `Action` и затем проводить откат транзакции. Так делается для того, чтобы

любые тесты создания/обновления/удаления оставляли базу данных в состоянии, в котором она пребывала до прогона теста. Поскольку класс ApplicationDbContext сконфигурирован с целью включения повторений при возникновении кратковременных ошибок, весь процесс обязан выполняться в соответствии со стратегией выполнения ApplicationDbContext.

Метод ExecuteInATransaction() выполняется с применением одиночного экземпляра ApplicationDbContext. Метод ExecuteInASharedTransaction() позволяет нескольким экземплярам ApplicationDbContext совместно использовать транзакцию. Вы узнаете больше об упомянутых методах позже в главе, а пока добавьте в свой класс BaseTest следующий код:

```
protected void ExecuteInATransaction(Action actionToExecute)
{
    var strategy = Context.Database.CreateExecutionStrategy();
    strategy.Execute(() =>
    {
        using var trans = Context.Database.BeginTransaction();
        actionToExecute();
        trans.Rollback();
    });
}
protected void ExecuteInASharedTransaction(Action<IDbContextTransaction> actionToExecute)
{
    var strategy = Context.Database.CreateExecutionStrategy();
    strategy.Execute(() =>
    {
        using IDbContextTransaction trans =
            Context.Database.BeginTransaction(IsolationLevel.ReadUncommitted);
        actionToExecute(trans);
        trans.Rollback();
    });
}
```

Добавление класса тестовой оснастки EnsureAutoLotDatabase

Инфраструктура тестирования xUnit предоставляет механизм, который позволяет запускать код до прогона любого теста (называется настройкой оснастки) и после прогона всех тестов (называется освобождением оснастки). Обычно поступать так не рекомендуется, но в рассматриваемом случае желательно удостовериться, что база данных создана и загружена данными до прогона любого теста, а не до прогона каждого теста. Классы тестов, которые реализуют IClassFixture<T> where T: TestFixtureClass, должны будут выполнять код конструктора типа T (т.е. TestFixtureClass) до прогона любого теста и код метода Dispose() после завершения всех тестов.

Создайте в каталоге Base новый файл класса по имени EnsureAutoLotDatabase TestFixture.cs и реализуйте интерфейс IDisposable. Сделайте класс открытым и запечатанным, а также добавьте показанные далее операторы using:

```
using System;
using AutoLot.Dal.Initialization;
```

```
namespace AutoLot.Dal.Tests.Base
{
    public sealed class EnsureAutoLotDatabaseTestFixture : IDisposable
    {
        ...
    }
}
```

В конструкторе понадобится создать экземпляр реализации IConfiguration и с его помощью создать экземпляр ApplicationDbContext. Затем нужно вызывать метод ClearAndReseedDatabase() класса SampleDataInitializer и в заключение освободить экземпляр контекста. В приводимых здесь примерах метод Dispose() не обязан выполнять какую-то работу (но должен присутствовать для соответствия шаблону с интерфейсом IDisposable). Вот как выглядит конструктор и метод Dispose():

```
public EnsureAutoLotDatabaseTestFixture()
{
    var configuration = TestHelpers.GetConfiguration();
    var context = TestHelpers.GetContext(configuration);
    SampleDataInitializer.ClearAndReseedDatabase(context);
    context.Dispose();
}

public void Dispose()
{
}
```

Добавление классов интеграционных тестов

Теперь необходимо создать классы, которые будут поддерживать автоматизированные тесты. Такие классы называют *тестовыми оснастками*. Добавьте в проект AutoLot.Dal.Tests новый каталог по имени IntegrationTests и поместите в него четыре файла с именами CarTests.cs, CustomerTests.cs, MakeTests.cs и OrderTests.cs.

В зависимости от возможностей средства запуска тестов тесты xUnit выполняются последовательно внутри тестовой оснастки (класса), но параллельно во всех тестовых оснастках (классах). Это может оказаться проблематичным при прогоне интеграционных тестов, взаимодействующих с единственной базой данных. Выполнение можно сделать последовательным для всех тестовых оснасток, добавив их в одну и ту же тестовую коллекцию. Тестовые коллекции определяются по имени с применением атрибута [Collection] к классу. Поместите перед всеми четырьмя классами следующий атрибут [Collection]:

```
[Collection("Integration Tests")]
```

Унаследуйте все четыре класса от BaseTest, реализуйте интерфейс IClassFixture и приведите операторы using к показанному далее виду:

```
// CarTests.cs
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.Exceptions;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Tests.Base;
using AutoLot.Models.Entities;
using Microsoft.EntityFrameworkCore;
```

```
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore.Query;
using Microsoft.EntityFrameworkCore.Storage;
using Xunit;
namespace AutoLot.Dal.Tests.IntegrationTests
{
    [Collection("Integation Tests")]
    public class CarTests : BaseTest,
        IClassFixture<EnsureAutoLotDatabaseTestFixture>
    {
    }
}

// CustomerTests.cs
using System.Collections.Generic;
using System;
using System.Linq;
using System.Linq.Expressions;
using AutoLot.Dal.Tests.Base;
using AutoLot.Models.Entities;
using Microsoft.EntityFrameworkCore;
using Xunit;
namespace AutoLot.Dal.Tests.IntegrationTests
{
    [Collection("Integation Tests")]
    public class CustomerTests : BaseTest,
        IClassFixture<EnsureAutoLotDatabaseTestFixture>
    {
    }
}

// MakeTests.cs
using System.Linq;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Dal.Tests.Base;
using AutoLot.Models.Entities;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Xunit;
namespace AutoLot.Dal.Tests.IntegrationTests
{
    [Collection("Integation Tests")]
    public class MakeTests : BaseTest,
        IClassFixture<EnsureAutoLotDatabaseTestFixture>
    {
    }
}

// OrderTests.cs
using System.Linq;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Dal.Tests.Base;
using Microsoft.EntityFrameworkCore;
```

```
using Xunit;
namespace AutoLot.Dal.Tests.IntegrationTests
{
    [Collection("Integration Tests")]
    public class OrderTests : BaseTest,
        IClassFixture<EnsureAutoLotDatabaseTestFixture>
    {
    }
}
```

Добавьте в класс MakeTests конструктор, который создает экземпляр MakeRepo и присваивает его закрытой переменной readonly уровня класса. Переопределите метод Dispose() и освободите в нем экземпляр MakeRepo:

```
[Collection("Integration Tests")]
public class MakeTests : BaseTest,
    IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
    private readonly IMakeRepo _repo;
    public MakeTests()
    {
        _repo = new MakeRepo(Context);
    }
    public override void Dispose()
    {
        _repo.Dispose();
    }
    ...
}
```

Повторите те же действия для класса OrderTests, но с использованием OrderRepo вместо MakeRepo:

```
[Collection("Integration Tests")]
public class OrderTests : BaseTest, IClassFixture<EnsureAutoLotDatabase
TestFixture>
{
    private readonly IOrderRepo _repo;
    public OrderTests()
    {
        _repo = new OrderRepo(Context);
    }
    public override void Dispose()
    {
        _repo.Dispose();
    }
    ...
}
```

Тестовые методы [*Fact*] и [*Theory*]

Тестовые методы без параметров называются *фактами* (и действуют атрибут [Fact]). Тестовые методы, которые принимают параметры, называются *теориями* (они используют атрибут [Theory]) и могут выполнять множество итераций с разными значениями, передаваемыми в качестве параметров. Чтобы взглянуть на такие

виды тестов, создайте в проекте AutoLot.Dal.Tests новый файл класса по имени SampleTests.cs. Вот как выглядит оператор using:

```
using Xunit;
namespace AutoLot.Dal.Tests
{
    public class SampleTests
    {
    }
}
```

Начните с создания теста [Fact]. В тесте [Fact] все значения содержатся внутри тестового метода. Следующий простой пример проверяет, что $3 + 2 = 5$:

```
[Fact]
public void SimpleFactTest()
{
    Assert.Equal(5, 3+2);
}
```

Что касается теста [Theory], то значения передаются тестовому методу и могут поступать из атрибута [InlineData], методов или классов. Здесь будет использоваться только атрибут [InlineData]. Создайте показанный ниже тест, которому предоставляются разные слагаемые и ожидаемый результат:

```
[Theory]
[InlineData(3, 2, 5)]
[InlineData(1, -1, 0)]
public void SimpleTheoryTest(int addend1, int addend2, int expectedResult)
{
    Assert.Equal(expectedResult, addend1+addend2);
}
```

На заметку! За дополнительными сведениями об инфраструктуре тестирования xUnit обращайтесь в документацию по ссылке <https://xunit.net/>.

Выполнение тестов

Хотя тесты xUnit можно запускать из командной строки (с применением dotnet test), разработчикам лучше использовать для этого Visual Studio. Выберите в меню Test (Тестирование) пункт Test Explorer (Проводник тестов), чтобы получить возможность прогонять и отлаживать все или выбранные тесты.

Запрашивание базы данных

Вспомните, что создание экземпляров сущностей из базы данных обычно предусматривает выполнение оператора LINQ в отношении свойств DbSet<T>. Поставщик баз данных и механизм трансляции LINQ преобразуют операторы LINQ в запросы SQL, с помощью которых из базы данных читаются соответствующие данные. Данные можно также загружать посредством метода FromSqlRaw() или FromSqlInterpolated() с применением низкоуровневых запросов SQL. Сущности, загружаемые в коллекции DbSet<T>, по умолчанию добавляются в ChangeTracker, но отслеживание можно отключать. Данные, загружаемые в коллекции DbSet<T> без ключей, никогда не отслеживаются.

Если связанные сущности уже загружены в DbSet<T>, тогда EF Core будет связывать новые экземпляры по навигационным свойствам. Например, если сущности Car загружаются в коллекцию DbSet<Car> и затем связанные сущности Order загружаются в коллекцию DbSet<Order> того же самого экземпляра ApplicationContext, то навигационное свойство Car.Orders будет возвращать связанные сущности без повторного запрашивания базы данных.

Многие демонстрируемые здесь методы имеют асинхронные версии. Синтаксис запросов LINQ структурно одинаков, поэтому будут использоваться только синхронные версии.

Состояние сущности

Когда сущность создается за счет чтения данных из базы, значение EntityState устанавливается в Unchanged.

Запросы LINQ

Тип коллекции DbSet<T> реализует (помимо прочих) интерфейс IQueryble<T>, что позволяет применять команды LINQ языка C# для создания запросов, извлекающих информацию из базы данных. Наряду с тем, что все операторы LINQ языка C# доступны для использования с типом коллекции DbSet<T>, некоторые операторы LINQ могут не поддерживаться поставщиком баз данных, а в EF Core появились дополнительные операторы LINQ. Неподдерживаемый оператор LINQ, который невозможно транслировать в язык запросов поставщика баз данных, приведет к генерации исключения времени выполнения, если только он не является последним в цепочке операторов LINQ. Когда неподдерживаемый оператор LINQ находится в самом конце цепочки LINQ, он выполнится на клиентской стороне (в коде C#).

На заметку! Настоящая книга не задумывалась как полный справочник по LINQ, так что в ней приводится не особо много примеров. С дополнительными примерами запросов LINQ можно ознакомиться по ссылке <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>.

Выполнение запросов LINQ

Вспомните, что при использовании LINQ для запрашивания из базы данных списка сущностей запрос не выполняется до тех пор, пока не начнется проход по результатам запроса, пока запрос не будет преобразован в List<T> (или объект) либо же пока не произойдет привязка запроса к списковому элементу управления (вроде сетки данных). Запрос единственной записи выполняется немедленно в случае применения вызова First(), Single() и т.д.

Нововведением версии EF Core 5 стало то, что в большинстве запросов LINQ можно вызывать метод ToQueryString() для исследования запроса, который выполняется в отношении базы данных. Для разделяемых запросов метод ToQueryString() возвращает только первый запрос, который будет выполняться. В рассматриваемых далее тестах это значение по возможности присваивается переменной (qs), чтобы вы могли изучить запрос во время отладки тестов.

Первый набор тестов (если только специально не указано иначе) находится в файле класса CustomerTests.cs.

Получение всех записей

Чтобы получить все записи из таблицы, просто используйте свойство `DbSet<T>` на-прямую без каких-либо операторов LINQ. Добавьте приведенный ниже тест [Fact]:

```
[Fact]
public void ShouldGetAllOfTheCustomers()
{
    var qs = Context.Customers.ToQueryString();
    var customers = Context.Customers.ToList();
    Assert.Equal(5, customers.Count);
}
```

Выделенный полужирным оператор транслируется в следующий код SQL:

```
SELECT [c].[Id], [c].[TimeStamp], [c].[FirstName],
       [c].[FullName], [c].[LastName] FROM [dbo].[Customers] AS [c]
```

Тот же самый процесс применяется для сущностей без ключей, подобных модели представления `CustomerOrderViewModel`, которая сконфигурирован на получение своих данных из представления `CustomerOrderView`:

```
modelBuilder.Entity<CustomerOrderViewModel>().HasNoKey()
    .ToView("CustomerOrderView", "dbo");
```

Экземпляр `DbSet<T>` для моделей представлений предлагает всю мощь запросов `DbSet<T>` для сущности с ключом. Отличие касается возможностей обновления. Изменения модели представления не могут быть сохранены в базе данных, тогда как изменения сущностей с ключами — могут. Добавьте в файл класса `OrderTest.cs` показанный далее тест, чтобы продемонстрировать получение данных из представления:

```
public void ShouldGetAllViewModels()
{
    var qs = Context.Orders.ToQueryString();
    var orders = Context.Orders.ToList();
    Assert.NotEmpty(orders);
    Assert.Equal(5, orders.Count);
}
```

Выделенный полужирным оператор транслируется в следующий код SQL:

```
SELECT [c].[Color], [c].[FirstName], [c].[IsDriveable],
       [c].[LastName], [c].[Make], [c].[PetName]
FROM [dbo].[CustomerOrderView] AS [c]
```

Фильтрация записей

Метод `Where()` используется для фильтрации записей из `DbSet<T>`. Несколько вызовов `Where()` можно плавно объединять в цепочку для динамического построения запроса. Выстроенные в цепочку вызовы `Where()` всегда объединяются с помощью операции “И”. Для объединения условий с применением операции “ИЛИ” необходимо использовать один вызов `Where()`.

Приведенный ниже тест возвращает заказчиков с фамилией, начинающейся с буквы “W” (нечувствительно к регистру символов):

234 Часть VII. Entity Framework Core

```
[Fact]
public void ShouldGetCustomersWithLastNameW()
{
    IQueryable<Customer> query = Context.Customers
        .Where(x => x.PersonalInformation.LastName.StartsWith("W"));
    var qs = query.ToQueryString();
    List<Customer> customers = query.ToList();
    Assert.Equal(2, customers.Count);
}
```

Запрос LINQ транслируется в следующий код SQL:

```
SELECT [c].[Id], [c].[TimeStamp], [c].[FirstName],
       [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE [c].[LastName] IS NOT NULL AND ([c].[LastName] LIKE N'W%')
```

Показанный далее тест возвращает заказчиков с фамилией, начинающейся с буквы "W" (нечувствительно к регистру символов), и именем, начинающимся с буквы "M" (нечувствительно к регистру символов), а также демонстрирует объединение вызовов Where() в цепочку в запросе LINQ:

```
[Fact]
public void ShouldGetCustomersWithLastNameWAndFirstNameM()
{
    IQueryable<Customer> query = Context.Customers
        .Where(x => x.PersonalInformation.LastName.StartsWith("W"))
        .Where(x => x.PersonalInformation.FirstName.StartsWith("M"));
    var qs = query.ToQueryString();
    List<Customer> customers = query.ToList();
    Assert.Single(customers);
}
```

Следующий тест возвращает заказчиков с фамилией, начинающейся с буквы "W" (нечувствительно к регистру символов), и именем, начинающимся с буквы "M" (нечувствительно к регистру символов), с применением единственного вызова Where():

```
[Fact]
public void ShouldGetCustomersWithLastNameWAndFirstNameM()
{
    IQueryable<Customer> query = Context.Customers
        .Where(x => x.PersonalInformation.LastName.StartsWith("W") &&
                   x.PersonalInformation.FirstName.StartsWith("M"));
    var qs = query.ToQueryString();
    List<Customer> customers = query.ToList();
    Assert.Single(customers);
}
```

Оба запроса транслируются в такой код SQL:

```
SELECT [c].[Id], [c].[TimeStamp], [c].[FirstName], [c].[FullName],
       [c].[LastName] FROM [Dbo].[Customers] AS [c]
WHERE ([c].[LastName] IS NOT NULL AND ([c].[LastName] LIKE N'W%'))
AND ([c].[FirstName] IS NOT NULL AND ([c].[FirstName] LIKE N'M%'))
```

Приведенный ниже тест возвращает заказчиков с фамилией, начинающейся с буквы "W" (нечувствительно к регистру символов), или именем, начинающимся с буквы "H" (нечувствительно к регистру символов):

```
[Fact]
public void ShouldGetCustomersWithLastNameWOrH()
{
    IQueryable<Customer> query = Context.Customers
        .Where(x => x.PersonalInformation.LastName.StartsWith("W") ||
                    x.PersonalInformation.LastName.StartsWith("H"));
    var qs = query.ToQueryString();
    List<Customer> customers = query.ToList();
    Assert.Equal(3, customers.Count);
}
```

Запрос LINQ транслируется в следующий код SQL:

```
SELECT [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE ([c].[LastName] IS NOT NULL AND ([c].[LastName] LIKE N'W%')) OR ([c].[LastName] IS NOT NULL AND ([c].[LastName] LIKE N'H%'))
```

Показанный далее тест возвращает заказчиков с фамилией, начинающейся с буквы "W" (нечувствительно к регистру символов), или именем, начинающимся с буквы "H" (нечувствительно к регистру символов), и демонстрирует использование метода EF.Functions.Like(). Обратите внимание, что включать групповой символ (%) вы должны самостоятельно.

```
[Fact]
public void ShouldGetCustomersWithLastNameWOrH()
{
    IQueryable<Customer> query = Context.Customers
        .Where(x => EF.Functions.Like(x.PersonalInformation.LastName, "W%") ||
                    EF.Functions.Like(x.PersonalInformation.LastName, "H%"));
    var qs = query.ToQueryString();
    List<Customer> customers = query.ToList();
    Assert.Equal(3, customers.Count);
}
```

Запрос LINQ транслируется в следующий код SQL (обратите внимание, что проверка на null не делается):

```
SELECT [c].[Id], [c].[TimeStamp], [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE ([c].[LastName] LIKE N'W%') OR ([c].[LastName] LIKE N'H%')
```

В приведенном ниже тесте из класса CarTests.cs применяется [Theory] для проверки количества записей Car в таблице Inventory на основе MakeId (метод IgnoreQueryFilters() рассматривался в разделе "Глобальные фильтры запросов" главы 22):

```
[Theory]
[InlineData(1, 2)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCarsByMake(int makeId, int expectedCount)
{
```

```
IQueryable<Car> query =
    Context.Cars.IgnoreQueryFilters().Where(x => x.MakeId == makeId);
var qs = query.ToQueryString();
var cars = query.ToList();
Assert.Equal(expectedCount, cars.Count);
}
```

Каждая строка [InlineData] становится уникальным тестом в средстве запуска тестов. В этом примере обрабатываются шесть тестов и в отношении базы данных выполняются шесть запросов. Вот как выглядит код SQL для одного из тестов (единственным отличием в запросах для других тестов в [Theory] будет значение MakeId):

```
DECLARE @_makeId_0 int = 1;
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
WHERE [i].[MakeId] = @_makeId_0
```

Следующий тест [Theory] показывает фильтрованный запрос с CustomerOrderViewModel (поместите тест в файл класса OrderTests.cs):

```
[Theory]
[InlineData("Black", 2)]
[InlineData("Rust", 1)]
[InlineData("Yellow", 1)]
[InlineData("Green", 0)]
[InlineData("Pink", 1)]
[InlineData("Brown", 0)]

public void ShouldGetAllViewModelsByColor(string color, int expectedCount)
{
    var query = _repo.GetOrdersViewModel().Where(x=>x.Color == color);
    var qs = query.ToQueryString();
    var orders = query.ToList();
    Assert.Equal(expectedCount, orders.Count);
}
```

Для первого теста [InlineData] генерируется такой запрос:

```
DECLARE @_color_0 nvarchar(4000) = N'Black';
SELECT [c].[Color], [c].[FirstName], [c].[IsDriveable],
       [c].[LastName], [c].[Make], [c].
       [PetName]
FROM [dbo].[CustomerOrderView] AS [c]
WHERE [c].[Color] = @_color_0
```

Сортировка записей

Методы OrderBy() и OrderByDescending() устанавливают для запроса сортировку (сортировки) по возрастанию и по убыванию. Если требуются дальнейшие сортировки, тогда используйте методы ThenBy() и ThenByDescending(). Сортировка демонстрируется в teste ниже:

```
[Fact]
public void ShouldSortByLastNameThenFirstName()
{
```

```
// Сортировать по фамилии, затем по имени.
var query = Context.Customers
    .OrderBy(x => x.PersonalInformation.LastName)
    .ThenBy(x => x.PersonalInformation.FirstName);
var qs = query.ToQueryString();
var customers = query.ToList();

// Если есть только один пользователь, то проверять нечего.
if (customers.Count <= 1) { return; }
for (int x = 0; x < customers.Count - 1; x++)
{
    var pi = customers[x].PersonalInformation;
    var pi2 = customers[x + 1].PersonalInformation;
    var compareLastName = string.Compare(pi.LastName,
        pi2.LastName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareLastName <= 0);
    if (compareLastName != 0) continue;
    var compareFirstName = string.Compare(pi.FirstName,
        pi2.FirstName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareFirstName <= 0);
}
}
```

Предыдущий запрос LINQ транслируется следующим образом:

```
SELECT [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
ORDER BY [c].[LastName], [c].[FirstName]
```

Сортировка записей в обратном порядке

Метод `Reverse()` меняет порядок сортировки на противоположный, как видно в представленном далее тесте:

```
[Fact]
public void ShouldSortByFirstNameThenLastNameUsingReverse()
{
    // Сортировать по фамилии, затем по имени,
    // и изменить порядок сортировки на противоположный.
    // Sort by Last name then first name then reverse the sort.
    var query = Context.Customers
        .OrderBy(x => x.PersonalInformation.LastName)
        .ThenBy(x => x.PersonalInformation.FirstName)
        .Reverse();
    var qs = query.ToQueryString();
    var customers = query.ToList();

    // Если есть только один пользователь, то проверять нечего.
    if (customers.Count <= 1) { return; }
    for (int x = 0; x < customers.Count - 1; x++)
    {
        var pil = customers[x].PersonalInformation;
        var pi2 = customers[x + 1].PersonalInformation;
        var compareLastName = string.Compare(pil.LastName,
            pi2.LastName, StringComparison.CurrentCultureIgnoreCase);
```

```

        Assert.True(compareLastName >= 0);
        if (compareLastName != 0) continue;
        var compareFirstName = string.Compare(pi1.FirstName,
            pi2.FirstName, StringComparison.CurrentCultureIgnoreCase);
        Assert.True(compareFirstName >= 0);
    }
}

```

Вот во что транслируется предыдущий запрос LINQ:

```

SELECT [c].[Id], [c].[TimeStamp],
    [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [dbo].[Customers] AS [c]
ORDER BY [c].[LastName] DESC, [c].[FirstName] DESC

```

Извлечение одиночной записи

Существуют три главных метода для возвращения одиночной записи посредством запроса: `First()`/`FirstOrDefault()`, `Last()`/`LastOrDefault()` и `Single()`/`SingleOrDefault()`. Хотя все они возвращают одиночную запись, принятые в них подходы отличаются. Методы и их варианты более подробно описаны ниже.

- Метод `First()` возвращает первую запись, которая соответствует условию запроса и любым конструкциям упорядочения. Если конструкции упорядочения не указаны, то возвращаемая запись основывается на порядке, установленном в базе данных. Если запись не возвращается, тогда генерируется исключение.
- Поведение метода `FirstOrDefault()` совпадает с поведением `First()`, но при отсутствии записей, соответствующих запросу, `FirstOrDefault()` возвращает стандартное значение для типа (`null`).
- Метод `Single()` возвращает первую запись, которая соответствует условию запроса и любым конструкциям упорядочения. Если конструкции упорядочения не указаны, то возвращаемая запись основывается на порядке, установленном в базе данных. Если запросу не соответствует одна или большее число записей, тогда генерируется исключение.
- Поведение метода `SingleOrDefault()` совпадает с поведением `Single()`, но при отсутствии записей, соответствующих запросу, `SingleOrDefault()` возвращает стандартное значение для типа (`null`).
- Метод `Last()` возвращает последнюю запись, которая соответствует условию запроса и любым конструкциям упорядочения. Если конструкции упорядочения не указаны, то возвращаемая запись основывается на порядке, установленном в базе данных. Если запись не возвращается, тогда генерируется исключение.
- Поведение метода `LastOrDefault()` совпадает с поведением `Last()`, но при отсутствии записей, соответствующих запросу, `LastOrDefault()` возвращает стандартное значение для типа (`null`).

Все методы могут также принимать `Expression<Func<T, bool>>` (лямбда-выражение) для фильтрации результирующего набора. Это означает, что вы можете поместить выражение `Where()` внутрь вызова `First()`/`Single()`. Следующие операторы эквивалентны:

```

Context.Customers.Where(c=>c.Id < 5).First();
Context.Customers.First(c=>c.Id < 5);

```

Из-за немедленного выполнения операторов LINQ, извлекающих одиночную запись, метод `ToQueryString()` оказывается недоступным. Приводимые трансляции запросов в код SQL получены с применением профилировщика SQL Server.

Использование `First() / FirstOrDefault()`

При использовании формы `First()` и `FirstOrDefault()` без параметров будет возвращаться первая запись (на основе порядка в базе данных или предшествующих конструкций упорядочения).

Показанный далее тест получает первую запись на основе порядка в базе данных:

```
[Fact]
public void GetFirstMatchingRecordDatabaseOrder()
{
    // Получить первую запись на основе порядка в базе данных.
    var customer = Context.Customers.First();
    Assert.Equal(1, customer.Id);
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
```

Следующий тест получает первую запись на основе порядка "фамилия, имя":

```
[Fact]
public void GetFirstMatchingRecordNameOrder()
{
    // Получить первую запись на основе порядка "фамилия, имя".
    var customer = Context.Customers
        .OrderBy(x => x.PersonalInformation.LastName)
        .ThenBy(x => x.PersonalInformation.FirstName)
        .First();
    Assert.Equal(1, customer.Id);
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
ORDER BY [c].[LastName], [c].[FirstName]
```

Приведенный ниже тест выдвигает утверждение о том, что если для `First()` не найдено соответствие, тогда генерируется исключение:

```
[Fact]
public void FirstShouldThrowExceptionIfNoneMatch()
{
    // Фильтровать на основе Id.
    // Сгенерировать исключение, если соответствие не найдено.
    Assert.Throws<InvalidOperationException>(() => Context.Customers.First(x => x.Id == 10));
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE [c].[Id] = 10
```

На заметку! `Assert.Throws()` — это специальный тип утверждения, который ожидает, что код в выражении сгенерирует исключение. Если исключение не было сгенерировано, тогда утверждение терпит неудачу.

В случае применения метода `FirstOrDefault()`, если соответствие не найдено, то результатом будет `null`, а не исключение:

```
[Fact]
public void FirstOrDefaultShouldReturnDefaultIfNoneMatch()
{
    // Expression<Func<Customer>> - это лямбда-выражение.
    Expression<Func<Customer, bool>> expression = x => x.Id == 10;

    // Возвращает null, если ничего не найдено.
    var customer = Context.Customers.FirstOrDefault(expression);
    Assert.Null(customer);
}
```

Предыдущий запрос LINQ транслируется в тот же код SQL, что и ранее:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE [c].[Id] = 10
```

Использование `Last()`/`LastOrDefault()`

При использовании формы `Last()` и `LastOrDefault()` без параметров будет возвращаться последняя запись (на основе предшествующих конструкций упорядочения). Показанный далее тест получает последнюю запись на основе порядка “фамилия, имя”:

```
[Fact]
public void GetLastMatchingRecordNameOrder()
{
    // Получить последнюю запись на основе порядка "фамилия, имя".
    var customer = Context.Customers
        .OrderBy(x => x.PersonalInformation.LastName)
        .ThenBy(x => x.PersonalInformation.FirstName)
        .Last();
    Assert.Equal(4, customer.Id);
}
```

Инфраструктура EF Core инвертирует операторы ORDER BY и затем получает результат с помощью TOP(1). Вот как выглядит выполняемый запрос:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
ORDER BY [c].[LastName] DESC, [c].[FirstName] DESC
```

Использование Single() / SingleOrDefault()

Концептуально Single() / SingleOrDefault() работает аналогично First() / FirstOrDefault(). Основное отличие в том, что метод Single() / SingleOrDefault() возвращает TOP(2), а не TOP(1), и генерирует исключение, если из базы данных возвращаются две записи. Следующий тест извлекает одиночную запись, в которой значение Id равно 1:

```
[Fact]
public void GetOneMatchingRecordWithSingle()
{
    // Получить первую запись на основе порядка в базе данных.
    var customer = Context.Customers.Single(x => x.Id == 1);
    Assert.Equal(1, customer.Id);
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(2) [c].[Id], [c].[TimeStamp],
    [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [dbo].[Customers] AS [c]
WHERE [c].[Id] = 1
```

Если запись не возвращается, тогда метод Single() генерирует исключение:

```
[Fact]
public void SingleShouldThrowExceptionIfNoneMatch()
{
    // Фильтровать на основе Id.
    // Сгенерировать исключение, если соответствие не найдено.
    Assert.Throws<InvalidOperationException>(() 
        => Context.Customers.Single(x => x.Id == 10));
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(2) [c].[Id], [c].[TimeStamp],
    [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [dbo].[Customers] AS [c]
WHERE [c].[Id] = 10
```

Если при использовании Single() или SingleOrDefault() возвращается больше чем одна запись, тогда генерируется исключение:

```
[Fact]
public void SingleShouldThrowExceptionIfMoreThanOneMatch()
{
    // Сгенерировать исключение, если найдено более одного соответствия.
    Assert.Throws<InvalidOperationException>(() 
        => Context.Customers.Single());
}

[Fact]
public void SingleOrDefaultShouldThrowExceptionIfMoreThanOneMatch()
{
    // Сгенерировать исключение, если найдено более одного соответствия.
    Assert.Throws<InvalidOperationException>(() 
        => Context.Customers.SingleOrDefault());
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(2) [c].[Id], [c].[TimeStamp],
       [c].[FirstName], [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
```

Если никакие данные не возвращаются в случае применения `SingleOrDefault()`, то результатом будет `null`, а не исключение:

```
[Fact]
public void SingleOrDefaultShouldReturnDefaultIfNoneMatch()
{
    // Expression<Func<Customer>> - это лямбда-выражение.
    Expression<Func<Customer, bool>> expression = x => x.Id == 10;
    // Возвращается null, когда ничего не найдено.
    var customer = Context.Customers.SingleOrDefault(expression);
    Assert.Null(customer);
}
```

Предыдущий запрос LINQ транслируется в такой код SQL:

```
SELECT TOP(1) [c].[Id], [c].[TimeStamp], [c].[FirstName],
       [c].[FullName], [c].[LastName]
FROM [Dbo].[Customers] AS [c]
WHERE [c].[Id] = 10
```

Глобальные фильтры запросов

Вспомните о наличии для сущности `Car` глобального фильтра запросов, который отбрасывает данные об автомобилях со значением свойства `IsDriveable`, равным `false`:

```
modelBuilder.Entity<Car>(entity =>
{
    entity.HasQueryFilter(c => c.IsDriveable);
    ...
});
```

Откройте файл класса `CarTests.cs` и добавьте показанный далее тест (все тесты в последующих разделах находятся в `CarTests.cs`, если не указано иначе):

```
[Fact]
public void ShouldReturnDriveableCarsWithQueryFilterSet()
{
    IQueryables<Car> query = Context.Cars;
    var qs = query.ToQueryString();
    var cars = query.ToList();
    Assert.NotEmpty(cars);
    Assert.Equal(9, cars.Count);
}
```

Также вспомните, что в процессе инициализации данных были созданы 10 записей об автомобилях, из которых один установлен как неуправляемый. При запуске запроса применяется глобальный фильтр запросов и выполняется следующий код SQL:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
WHERE [i].[IsDriveable] = CAST(1 AS bit)
```

На заметку! Как вскоре будет показано, глобальные фильтры запросов также применяются при загрузке связанных сущностей и при использовании методов `FromSqlRaw()` и `FromSqlInterpolated()`.

Отключение глобальных фильтров запросов

Чтобы отключить глобальные фильтры запросов для сущностей в запросе, добавьте к запросу LINQ вызов метода `IgnoreQueryFilters()`. Он заблокирует все фильтры для всех сущностей в запросе. Если есть несколько сущностей с глобальными фильтрами запросов и некоторые фильтры сущностей нужны, тогда потребуется поместить их в методы `Where()` оператора LINQ. Добавьте в файл класса `CarTests.cs` приведенный ниже тест, который отключает фильтр запросов и возвращает все записи:

```
[Fact]
public void ShouldGetAllOfTheCars()
{
    IQueryables<Car> query = Context.Cars.IgnoreQueryFilters();
    var qs = query.ToQueryString();
    var cars = query.ToList();
    Assert.Equal(10, cars.Count);
}
```

Как и можно было ожидать, в сгенерированном коде SQL больше нет конструкции `WHERE`, устраниющей записи для неуправляемых автомобилей:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
```

Фильтры запросов для навигационных свойств

Помимо глобального фильтра запросов для сущности `Car` был добавлен фильтр запросов к свойству `CarNavigation` сущности `Order`:

```
modelBuilder.Entity<Order>().HasQueryFilter(e
    => e.CarNavigation!.IsDriveable);
```

Чтобы увидеть его в действии, добавьте в файл класса `OrderTests.cs` следующий тест:

```
[Fact]
public void ShouldGetAllOrdersExceptFiltered()
{
    var query = Context.Orders.AsQueryable();
    var qs = query.ToQueryString();
    var orders = query.ToList();
    Assert.NotEmpty(orders);
    Assert.Equal(4, orders.Count);
}
```

Вот сгенерированный код SQL:

```
SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp]
FROM [dbo].[Orders] AS [o]
INNER JOIN (
    SELECT [i].[Id], [i].[IsDriveable]
    FROM [dbo].[Inventory] AS [i]
    WHERE [i].[IsDriveable] = CAST(1 AS bit)\r\n) AS [t]
ON [o].[CarId] = [t].[Id]
WHERE [t].[IsDriveable] = CAST(1 AS bit)
```

Поскольку навигационное свойство `CarNavigation` является *обязательным*, механизм трансляции запросов использует конструкцию `INNER JOIN`, исключая записи `Order`, где `Car` соответствует неуправляемому автомобилю. Для возвращения всех записей добавьте в запрос LINQ вызов `IgnoreQueryFilters()`.

Энергичная загрузка связанных данных

В предыдущей главе объяснялось, что сущности, которые связаны через навигационные свойства, могут создаваться в одном запросе с применением энергичной загрузки. Метод `Include()` указывает соединение со связанный сущностью, а метод `ThenInclude()` используется для последующих соединений. Оба метода будут задействованы в рассматриваемых далее тестах. Как упоминалось ранее, когда методы `Include()`/`ThenInclude()` транслируются в SQL, для обязательных отношений применяется внутреннее соединение, а для необязательных — левое соединение.

Поместите в файл класса `CarTests.cs` следующий тест, чтобы продемонстрировать одиночный вызов `Include()`:

```
[Fact]
public void ShouldGetAllOfTheCarsWithMakes()
{
    IIIncludableQueryable<Car, Make?> query =
        Context.Cars.Include(c => c.MakeNavigation);
    var queryString = query.ToQueryString();
    var cars = query.ToList();
    Assert.Equal(9, cars.Count);
}
```

Тест добавляет к результатам свойство `MakeNavigation`, выполняя внутреннее соединение с помощью показанного ниже кода SQL. Обратите внимание, что глобальный фильтр запросов действует:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].
[TimeStamp],
[m].[Id], [m].[Name], [m].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
INNER JOIN [dbo].[Makes] AS [m] ON [i].[MakeId] = [m].[Id]
WHERE [i].[IsDriveable] = CAST(1 AS bit)
```

Во втором тесте используется два набора связанных данных. Первый — это получение информации `Make` (как и в предыдущем тесте), а второй — получение сущностей `Order` и затем присоединенных к ним сущностей `Customer`. Полный тест также отфильтровывает записи `Car`, для которых есть записи `Order`. Для необязательных отношений генерируются левые соединения:

```
[Fact]
public void ShouldGetCarsOnOrderWithRelatedProperties()
{
    IIIncludableQueryable<Car, Customer?> query = Context.Cars
        .Where(c => c.Orders.Any())
        .Include(c => c.MakeNavigation)
        .Include(c => c.Orders).ThenInclude(o => o.CustomerNavigation);
    var queryString = query.ToQueryString();
    var cars = query.ToList();
    Assert.Equal(4, cars.Count);
```

```

cars.ForEach(c =>
{
    Assert.NotNull(c.MakeNavigation);
    Assert.NotNull(c.Orders.ToList()[0].CustomerNavigation);
});
}

```

Вот сгенерированный запрос:

```

SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp], [m].[Id], [m].[Name],
       [m].[TimeStamp], [t0].[Id], [t0].[CarId], [t0].[CustomerId],
       [t0].[TimeStamp], [t0].[Id0], [t0].[TimeStamp0],
       [t0].[FirstName], [t0].[FullName], [t0].[LastName], [t0].[Id1]
FROM [dbo].[Inventory] AS [i]
INNER JOIN [dbo].[Makes] AS [m] ON [i].[MakeId]=[m].[Id]
LEFT JOIN(SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp],
          [c].[Id] AS [Id0], [c].[TimeStamp] AS [TimeStamp0],
          [c].[FirstName], [c].[FullName], [c].[LastName], [t].[Id] AS [Id1]
       FROM [dbo].[Orders] AS [o]
       INNER JOIN(SELECT [i0].[Id], [i0].[IsDriveable]
                  FROM [dbo].[Inventory] AS [i0]
                  WHERE [i0].[IsDriveable]=CAST(1 AS BIT)) AS [t] ON [o].[CarId]=[t].[Id]
       INNER JOIN [dbo].[Customers] AS [c] ON [o].[CustomerId]=[c].[Id]
WHERE [t].[IsDriveable]=CAST(1 AS BIT) AS [t0]
ON [i].[Id]=[t0].[CarId]
WHERE([i].[IsDriveable]=CAST(1 AS BIT))AND EXISTS (SELECT 1
       FROM [dbo].[Orders] AS [o0]
       INNER JOIN(SELECT [i1].[Id], [i1].[Color], [i1].[IsDriveable],
                  [i1].[MakeId], [i1].[PetName], [i1].[TimeStamp]
               FROM [dbo].[Inventory] AS [i1]
               WHERE [i1].[IsDriveable]=CAST(1 AS BIT)) AS [t1] ON [o0].[CarId]=[t1].[Id]
WHERE([t1].[IsDriveable]=CAST(1 AS BIT)) AS [t1] ON [o0].[CarId]=[t1].[Id]
AND([i].[Id]=[o0].[CarId]))
ORDER BY [i].[Id], [m].[Id], [t0].[Id], [t0].[Id1], [t0].[Id0];

```

Разделение запросов к связанным данным

Чем больше соединений добавляется в запрос LINQ, тем сложнее становится результатирующий запрос. В версии EF Core 5 появилась возможность выполнять сложные соединения как разделенные запросы. Детальное обсуждение ищите в предыдущей главе, но вкратце помещение в запрос LINQ вызова метода `AsSplitQuery()` инструктирует инфраструктуру EF Core о необходимости разделения одного обращения к базе данных на несколько обращений. В итоге может повыситься эффективность, но возникает риск несогласованности данных. Добавьте в тестовую оснастку приведенный далее тест:

```

[Fact]
public void ShouldGetCarsOnOrderWithRelatedPropertiesAsSplitQuery()
{

```

246 Часть VII. Entity Framework Core

```
IQueryable<Car> query = Context.Cars.Where(c => c.Orders.Any())
    .Include(c => c.MakeNavigation)
    .Include(c => c.Orders).ThenInclude(o => o.CustomerNavigation)
    .AsSplitQuery();
var cars = query.ToList();
Assert.Equal(4, cars.Count);
cars.ForEach(c =>
{
    Assert.NotNull(c.MakeNavigation);
    Assert.NotNull(c.Orders.ToList()[0].CustomerNavigation);
});
}
```

Метод `ToQueryString()` возвращает только первый запрос, поэтому последующие запросы были получены с применением профилировщика SQL Server:

```
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
       [i].[PetName], [i].[TimeStamp], [m].[Id], [m].[Name],
       [m].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
INNER JOIN [dbo].[Makes] AS [m] ON [i].[MakeId] = [m].[Id]
WHERE ([i].[IsDriveable] = CAST(1 AS bit)) AND EXISTS (
    SELECT 1
    FROM [Dbo].[Orders] AS [o]
    INNER JOIN (
        SELECT [i0].[Id], [i0].[Color], [i0].[IsDriveable], [i0].[MakeId],
               [i0].[PetName], [i0].[TimeStamp]
        FROM [dbo].[Inventory] AS [i0]
        WHERE [i0].[IsDriveable] = CAST(1 AS bit)
    ) AS [t] ON [o].[CarId] = [t].[Id]
    WHERE ([t].[IsDriveable] = CAST(1 AS bit)) AND ([i].[Id] = [o].[CarId]))
ORDER BY [i].[Id], [m].[Id]

SELECT [t0].[Id], [t0].[CarId], [t0].[CustomerId], [t0].[TimeStamp],
       [t0].[Id1], [t0].[TimeStamp1], [t0].[FirstName],
       [t0].[FullName], [t0].[LastName], [i].[Id], [m].[Id]
FROM [dbo].[Inventory] AS [i]
INNER JOIN [dbo].[Makes] AS [m] ON [i].[MakeId] = [m].[Id]
INNER JOIN (
    SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp],
           [c].[Id] AS [Id1], [c].[TimeStamp] AS [TimeStamp1],
           [c].[FirstName], [c].[FullName], [c].[LastName]
    FROM [Dbo].[Orders] AS [o]
    INNER JOIN (
        SELECT [i0].[Id], [i0].[IsDriveable]
        FROM [dbo].[Inventory] AS [i0]
        WHERE [i0].[IsDriveable] = CAST(1 AS bit)
    ) AS [t] ON [o].[CarId] = [t].[Id]
    INNER JOIN [Dbo].[Customers] AS [c] ON [o].[CustomerId] = [c].[Id]
    WHERE [t].[IsDriveable] = CAST(1 AS bit)
) AS [t0] ON [i].[Id] = [t0].[CarId]
WHERE ([i].[IsDriveable] = CAST(1 AS bit)) AND EXISTS (
    SELECT 1
    FROM [Dbo].[Orders] AS [o0]
    INNER JOIN (
```

```

SELECT [i1].[Id], [i1].[Color], [i1].[IsDrivable], [i1].[MakeId],
       [i1].[PetName], [i1].[TimeStamp]
  FROM [dbo].[Inventory] AS [i1]
 WHERE [i1].[IsDrivable] = CAST(1 AS bit)
    ) AS [t1] ON [o0].[CarId] = [t1].[Id]
 WHERE ([t1].[IsDrivable] = CAST(1 AS bit)) AND ([i].[Id] = [o0].[CarId]))
ORDER BY [i].[Id], [m].[Id]

```

Будете вы разделять свои запросы или нет, зависит от существующих бизнес-требований.

Фильтрация связанных данных

В версии EF Core 5 появилась возможность фильтрации при включении навигационных свойств типа коллекций. До выхода EF Core 5 единственным способом получения отфильтрованного списка для навигационного свойства типа коллекций было использование явной загрузки. Добавьте в `MakeTests.cs` следующий тест, который демонстрирует получение записей производителей, выпускающих автомобили желтого цвета:

```

[Fact]
public void ShouldGetAllMakesAndCarsThatAreYellow()
{
    var query = Context.Makes.IgnoreQueryFilters()
        .Include(x => x.Cars.Where(x => x.Color == "Yellow"));
    var qs = query.ToQueryString();
    var makes = query.ToList();
    Assert.NotNull(makes);
    Assert.NotEmpty(makes);
    Assert.NotEmpty(makes.Where(x => x.Cars.Any()));
    Assert.Empty(makes.First(m => m.Id == 1).Cars);
    Assert.Empty(makes.First(m => m.Id == 2).Cars);
    Assert.Empty(makes.First(m => m.Id == 3).Cars);
    Assert.Single(makes.First(m => m.Id == 4).Cars);
    Assert.Empty(makes.First(m => m.Id == 5).Cars);
}

```

Ниже показан генерированный код SQL:

```

SELECT [m].[Id], [m].[Name], [m].[TimeStamp], [t].[Id], [t].[Color],
       [t].[IsDrivable], [t].[MakeId], [t].[PetName], [t].[TimeStamp]
  FROM [dbo].[Makes] AS [m]
LEFT JOIN (
    SELECT [i].[Id], [i].[Color], [i].[IsDrivable], [i].[MakeId],
           [i].[PetName], [i].[TimeStamp]
  FROM [dbo].[Inventory] AS [i]
 WHERE [i].[Color] = N'Yellow') AS [t] ON [m].[Id] = [t].[MakeId]
ORDER BY [m].[Id], [t].[Id]

```

Изменение запроса на разделенный приводит к выдаче такого кода SQL (получен с использованием профилировщика SQL Server):

```

SELECT [m].[Id], [m].[Name], [m].[TimeStamp]
  FROM [dbo].[Makes] AS [m]
ORDER BY [m].[Id]

```

```

SELECT [t].[Id], [t].[Color], [t].[IsDriveable], [t].[MakeId],
       [t].[PetName], [t].[TimeStamp], [m].[Id]
FROM [dbo].[Makes] AS [m]
INNER JOIN (
    SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
           [i].[PetName], [i].[TimeStamp]
    FROM [dbo].[Inventory] AS [i]
    WHERE [i].[Color] = N'Yellow'
) AS [t] ON [m].[Id] = [t].[MakeId]
ORDER BY [m].[Id]

```

Явная загрузка связанных данных

Если связанные данные нужно загрузить сразу после того, как главная сущность была запрошена в память, то связанные сущности можно извлечь из базы данных с помощью последующих обращений к базе данных. Это запускается с применением метода `Entry()` класса, производного от `DbContext`. При загрузке сущностей на стороне “многие” отношения “один ко многим” используйте вызов метода `Collection()` на результате `Entry()`. Чтобы загрузить сущности на стороне “один” отношения “один ко многим” (или отношения “один к одному”), применяйте метод `Reference()`. Вызов метода `Query()` на результате `Collection()` или `Reference()` возвращает экземпляр реализации `IQueryable<T>`, который можно использовать для получения строки запроса (как видно в приводимых далее тестах) и для управления фильтрами запросов (как показано в следующем разделе). Чтобы выполнить запрос и загрузить запись (записи), вызовите метод `Load()` на результате метода `Collection()`, `Reference()` или `Query()`. Выполнение запроса начнется немедленно после вызова `Load()`.

Представленный ниже тест (из `CarTests.cs`) демонстрирует, каким образом загрузить связанные данные через навигационное свойство типа ссылки внутри сущности `Car`:

```

[Fact]
public void ShouldGetReferenceRelatedInformationExplicitly()
{
    var car = Context.Cars.First(x => x.Id == 1);
    Assert.Null(car.MakeNavigation);
    var query = Context.Entry(car)
        .Reference(c => c.MakeNavigation).Query();
    var qs = query.ToQueryString();
    query.Load();
    Assert.NotNull(car.MakeNavigation);
}

```

Вот сгенерированный код SQL:

```

DECLARE @_p_0 int = 1;
SELECT [m].[Id], [m].[Name], [m].[TimeStamp]
FROM [dbo].[Makes] AS [m]
WHERE [m].[Id] = @_p_0

```

В следующем teste показано, как загрузить связанные данные через навигационное свойство типа коллекции внутри сущности `Car`:

```

[Fact]
public void ShouldGetCollectionRelatedInformationExplicitly()
{
}

```

```

var car = Context.Cars.First(x => x.Id == 1);
Assert.Empty(car.Orders);
var query = Context.Entry(car).Collection(c => c.Orders).Query();
var qs = query.ToQueryString();
query.Load();
Assert.Single(car.Orders);
}

```

Сгенерированный код SQL выглядит так:

```

DECLARE @_p_0 int = 1;
SELECT [o].[Id], [o].[CarId], [o].[CustomerId], [o].[TimeStamp]
FROM [dbo].[Orders] AS [o]
INNER JOIN (
    SELECT [i].[Id], [i].[IsDriveable]
    FROM [dbo].[Inventory] AS [i]
    WHERE [i].[IsDriveable] = CAST(1 AS bit)
) AS [t] ON [o].[CarId] = [t].[Id]
WHERE ([t].[IsDriveable] = CAST(1 AS bit)) AND ([o].[CarId] = @_p_0)

```

Явная загрузка связанных данных с фильтрами запросов

Глобальные фильтры запросов активны не только при формировании запросов, генерируемых для энергичной загрузки связанных данных, но и при явной загрузке связанных данных. Добавьте (в MakeTests.cs) приведенный далее тест:

```

[Theory]
[InlineData(1,1)]
[InlineData(2,1)]
[InlineData(3,1)]
[InlineData(4,2)]
[InlineData(5,3)]
[InlineData(6,1)]
public void ShouldGetAllCarsForAMakeExplicitlyWithQueryFilters(
    int makeId, int carCount)
{
    var make = Context.Makes.First(x => x.Id == makeId);
    IQueryables<Car> query = Context.Entry(make).Collection(c => c.Cars)
        .Query();
    var qs = query.ToQueryString();
    query.Load();
    Assert.Equal(carCount, make.Cars.Count());
}

```

Этот тест похож на тест ShouldGetTheCarsByMake() из раздела “Фильтрация записей” ранее в главе. Однако вместо того, чтобы просто получить записи Car, которые имеют определенное значение MakeId, текущий тест сначала получает запись Make и затем явно загружает записи Car для находящейся в памяти записи Make. Ниже показан сгенерированный код SQL:

```

DECLARE @_p_0 int = 5;
SELECT [i].[Id], [i].[Color], [i].[IsDriveable], [i].[MakeId],
    [i].[PetName], [i].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
WHERE ([i].[IsDriveable] = CAST(1 AS bit)) AND ([i].[MakeId] = @_p_0)

```

Обратите внимание на то, что фильтр запросов по-прежнему применяется, хотя главной сущностью в запросе является запись Make. Для отключения фильтров запросов при явной загрузке записей вызовите `IgnoreQueryFilters()` в сочетании с методом `Query()`. Вот тест, который отключает фильтры запросов (находится в `MakeTests.cs`):

```
[Theory]
[InlineData(1, 2)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetAllCarsForAMakeExplicitly(int makeId, int carCount)
{
    var make = Context.Makes.First(x => x.Id == makeId);
    IQueryables<Car> query = Context.Entry(make).Collection(c => c.Cars)
        .Query().IgnoreQueryFilters();
    var qs = query.IgnoreQueryFilters().ToQueryString();
    query.Load();
    Assert.Equal(carCount, make.Cars.Count());
}
```

Выполнение запросов SQL с помощью LINQ

Если оператор LINQ для отдельного запроса слишком сложен или тестирование показывает, что производительность оказалась ниже, чем желаемая, тогда данные можно извлекать с использованием низкоуровневого оператора SQL через метод `FromSqlRaw()` или `FromSqlInterpolated()` класса `DbSet<T>`. Оператором SQL может быть встроенный оператор SELECT языка T-SQL, хранимая процедура или табличная функция. Если запрос является открытым (например, оператор T-SQL без завершающей точки с запятой), тогда операторы LINQ можно добавлять к вызову метода `FromSqlRaw() / FromSqlInterpolated()` для дальнейшего определения генерируемого запроса. Полный запрос выполняется на серверной стороне с объединением оператора SQL и кода SQL, сгенерированного операторами LINQ.

Если оператор завершен или содержит код SQL, который не может быть достроен (скажем, задействует общие табличные выражения), то такой запрос все равно выполняется на серверной стороне, но любая дополнительная фильтрация и обработка должна делаться на клиентской стороне как LINQ to Objects. Метод `FromSqlRaw()` выполняет запрос в том виде, в котором он набран. Метод `FromSqlInterpolated()` применяет интерполяцию строк C# и помещает интерполированные значения в параметры. В следующих тестах (из `CarTests.cs`) демонстрируются примеры использования обоих методов с глобальными фильтрами запросов и без них:

```
[Fact]
public void ShouldNotGetTheLemonsUsingFromSql()
{
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
    var cars = Context.Cars.FromSqlRaw($"Select * from {schemaName}.{tableName}")
        .ToList();
    Assert.Equal(9, cars.Count());
}
```

```
[Fact]
public void ShouldGetTheCarsUsingFromSqlWithIgnoreQueryFilters()
{
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
    var cars = Context.Cars.FromSqlRaw($"Select * from {schemaName}.{tableName}")
        .IgnoreQueryFilters().ToList();
    Assert.Equal(10, cars.Count);
}

[Fact]
public void ShouldGetOneCarUsingInterpolation()
{
    var carId = 1;
    var car = Context.Cars
        .FromSqlInterpolated($"Select * from dbo.Inventory where Id = {carId}")
        .Include(x => x.MakeNavigation)
        .First();
    Assert.Equal("Black", car.Color);
    Assert.Equal("VW", car.MakeNavigation.Name);
}

[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCarsByMakeUsingFromSql(int makeId,
                                                int expectedCount)
{
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
    var cars = Context.Cars.FromSqlRaw($"Select * from {schemaName}.{tableName}")
        .Where(x => x.MakeId == makeId).ToList();
    Assert.Equal(expectedCount, cars.Count);
}
```

Во время применения методов `FromSqlRaw()`/`FromSqlInterpolated()` действует ряд правил: столбцы, возвращаемые из оператора SQL, должны соответствовать столбцам в модели, должны возвращаться все столбцы для модели, а возвращать связанные данные не допускается.

Методы агрегирования

В EF Core также поддерживаются методы агрегирования серверной стороны (`Max()`, `Min()`, `Count()`, `Average()` и т.д.). Вызовы методов агрегирования можно добавлять в конец запроса LINQ с вызовами `Where()` или же сам вызов метода агрегирования может содержать выражение фильтра (подобно `First()` и `Single()`). Агрегирование выполняется на серверной стороне и из запроса возвращается одиночное значение. Глобальные фильтры запросов оказывают воздействие на методы агрегирования и могут быть отключены с помощью `IgnoreQueryFilters()`.

252 Часть VII. Entity Framework Core

Все операторы SQL, показанные в этом разделе, были получены с использованием профилировщика SQL Server.

Первый тест (из CarTests.cs) просто подсчитывает все записи Car в базе данных. Из-за того, что фильтр запросов активен, результатом подсчета будет 9:

```
[Fact]
public void ShouldGetTheCountOfCars()
{
    var count = Context.Cars.Count();
    Assert.Equal(9, count);
}
```

Ниже приведен код SQL, который выполнялся:

```
SELECT COUNT(*)
FROM [dbo].[Inventory] AS [i]
WHERE [i].[IsDriveable] = CAST(1 AS bit)
```

После добавления вызова IgnoreQueryFilters() метод Count() возвращает 10 и конструкция WHERE удаляется из запроса SQL:

```
[Fact]
public void ShouldGetTheCountOfCarsIgnoreQueryFilters()
{
    var count = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(10, count);
}
```

Вот сгенерированный код SQL:

```
SELECT COUNT(*) FROM [dbo].[Inventory] AS [i]
```

Следующие тесты (из CarTests.cs) демонстрируют метод Count() с условием WHERE. В первом тесте выражение добавляется прямо в вызов метода Count(), а во втором вызов метода Count() помещается в конец запроса LINQ:

```
[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]

public void ShouldGetTheCountOfCarsByMakeP1(int makeId, int expectedCount)
{
    var count = Context.Cars.Count(x=>x.MakeId == makeId);
    Assert.Equal(expectedCount, count);
}

[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
```

```
public void ShouldGetTheCountOfCarsByMakeP2(int makeId, int expectedCount)
{
    var count = Context.Cars.Where(x => x.MakeId == makeId).Count();
    Assert.Equal(expectedCount, count);
}
```

Оба теста создают те же самые обращения SQL к серверу (в каждом тесте значение для MakeId изменяется на основе [InlineData]):

```
exec sp_executesql N'SELECT COUNT(*)
FROM [dbo].[Inventory] AS [i]
WHERE ([i].[IsDriveable] = CAST(1 AS bit)) AND ([i].[MakeId] = @_makeId_0)
',N'@_makeId_0 int', @_makeId_0=6
```

Any() и All()

Методы Any() и All() проверяют набор записей, чтобы выяснить, соответствует ли критериям любая запись (Any()) или же все записи (All()). Как и вызовы методов агрегирования, их можно добавлять в конец запроса LINQ с вызовами Where() либо же помещать выражение фильтрации в сам вызов метода. Методы Any() и All() выполняются на серверной стороне, а из запроса возвращается булевское значение. Глобальные фильтры запросов оказывают воздействие на методы Any() и All(); их можно отключить с помощью IgnoreQueryFilters().

Все операторы SQL, показанные в этом разделе, были получены с применением профилировщика SQL Server. Первый тест (из CarTests.cs) проверяет, имеет ли любая запись Car специфическое значение MakeId:

```
[Theory]
[InlineData(1, true)]
[InlineData(11, false)]
public void ShouldCheckForAnyCarsWithMake(int makeId, bool expectedResult)
{
    var result = Context.Cars.Any(x => x.MakeId == makeId);
    Assert.Equal(expectedResult, result);
}
```

Для первого теста [Theory] выполняется следующий код SQL:

```
exec sp_executesql N'SELECT CASE
WHEN EXISTS (
    SELECT 1
    FROM [dbo].[Inventory] AS [i]
    WHERE ([i].[IsDriveable] = CAST(1 AS bit))
        AND ([i].[MakeId] = @_makeId_0)) THEN
    CAST(1 AS bit)
ELSE CAST(0 AS bit)
END',N'@_makeId_0 int', @_makeId_0=1
```

Второй тест проверяет, имеют ли все записи Car специфическое значение MakeId:

```
[Theory]
[InlineData(1, false)]
[InlineData(11, false)]
public void ShouldCheckForAllCarsWithMake(int makeId, bool expectedResult)
{
```

254 Часть VII. Entity Framework Core

```
var result = Context.Cars.All(x => x.MakeId == makeId);
Assert.Equal(expectedResult, result);
}
```

Вот код SQL, выполняемый для второго теста [Theory]:

```
exec sp_executesql N'SELECT CASE
WHEN NOT EXISTS (
    SELECT 1
    FROM [dbo].[Inventory] AS [i]
    WHERE ([i].[IsDriveable] = CAST(1 AS bit))
        AND ([i].[MakeId] <> @_makeId_0)) THEN
    CAST(1 AS bit)
ELSE CAST(0 AS bit)
END',N'@_makeId_0 int', @_makeId_0=1
```

Получение данных из хранимых процедур

Последний шаблон извлечения данных, который необходимо изучить, предусматривает получение данных из хранимых процедур. Несмотря на некоторые проблемы EF Core в плане работы с хранимыми процедурами (по сравнению с EF 6), не забывайте, что инфраструктура EF Core построена поверх ADO.NET. Нужно просто спуститься на уровень ниже и вспомнить, как вызывались хранимые процедуры до появления инструментов объектно-реляционного отображения. Показанный далее метод в CarRepo создает обязательные параметры (входной и выходной), задействует свойство Database экземпляра ApplicationContext и вызывает ExecuteSqlRaw():

```
public string GetPetName(int id)
{
    var parameterId = new SqlParameter
    {
        ParameterName = "@carId",
        SqlDbType = System.Data.SqlDbType.Int,
        Value = id,
    };
    var parameterName = new SqlParameter
    {
        ParameterName = "@petName",
        SqlDbType = System.Data.SqlDbType.NVarChar,
        Size = 50,
        Direction = ParameterDirection.Output
    };
    var result = Context.Database
        .ExecuteSqlRaw("EXEC [dbo].[GetPetName] @carId, @petName OUTPUT",
                      parameterId, parameterName);
    return (string)parameterName.Value;
}
```

При наличии такого кода тест становится тривиальным. Добавьте в файл класса CarTests.cs следующий тест:

```
[Theory]
[InlineData(1, "Zippy")]
[InlineData(2, "Rusty")]
[InlineData(3, "Mel")]
```

```
[InlineData(4, "Clunker")]
[InlineData(5, "Bimmer")]
[InlineData(6, "Hank")]
[InlineData(7, "Pinky")]
[InlineData(8, "Pete")]
[InlineData(9, "Brownie")]
public void ShouldGetValueFromStoredProcedure(int id, string expectedName)
{
    Assert.Equal(expectedName, new CarRepo(Context).GetPetName(id));
}
```

Создание записей

Записи добавляются в базу данных за счет их создания в коде, добавления к `DbSet<T>` и вызова метода `SaveChanges()`/`SaveChangesAsync()` контекста. Во время выполнения метода `SaveChanges()` объект `ChangeTracker` сообщает обо всех добавленных сущностях, а инфраструктура EF Core вместе с поставщиком баз данных создают подходящий оператор (операторы) SQL для вставки записи (записей).

Вспомните, что метод `SaveChanges()` выполняется в неявной транзакции, если только не применяется явная транзакция. Если сохранение было успешным, то затем запрашиваются значения, сгенерированные сервером, для установки значений в сущностях. Все эти тесты будут использовать явную транзакцию, так что можно производить откат, оставив базу данных в том же состоянии, в каком она находилась, когда начинался прогон тестов.

Все операторы SQL, показанные далее в разделе, были получены с применением профилировщика SQL Server.

На заметку! Записи можно добавлять также с использованием класса, производного от `DbContext`. Во всех примерах для добавления записей будут применяться свойства `DbSet<T>`. В классах `DbSet<T>` и `DbContext` имеются асинхронные версии методов `Add()`/`AddRange()`, но здесь рассматриваются только синхронные версии.

Состояние сущности

Когда сущность создана с помощью кода, но еще не была добавлена в `DbSet<T>`, значением `EntityState` является `Detached`. После добавления новой сущности в `DbSet<T>` значение `EntityState` устанавливается в `Added`. В случае успешного выполнения `SaveChanges()` значение `EntityState` устанавливается в `Unchanged`.

Добавление одной записи

В следующем teste демонстрируется добавление одиночной записи в таблицу `Inventory`:

```
[Fact]
public void ShouldAddACar()
{
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
        var car = new Car
        {
```

```

        Color = "Yellow",
        MakeId = 1,
        PetName = "Herbie"
    };
    var carCount = Context.Cars.Count();
    Context.Cars.Add(car);
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    Assert.Equal(carCount+1,newCarCount);
}
}

```

Ниже приведен выполняемый оператор SQL. Обратите внимание, что у недавно добавленной сущности запрашиваются свойства, сгенерированные базой данных (Id и TimeStamp). Когда результат запроса поступает в исполняющую среду EF Core, сущность обновляется с использованием значений серверной стороны:

```

exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [dbo].[Inventory] ([Color], [MakeId], [PetName])
VALUES (@p0, @p1, @p2);
SELECT [Id], [IsDriveable], [TimeStamp]
FROM [dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
',N'@p0 nvarchar(50),@p1 int,@p2 nvarchar(50)',@p0=N'Yellow',@p1=1,
@p2=N'Herbie'

```

Добавление одной записи с использованием метода Attach()

Когда первичный ключ сущности сопоставлен со столбцом идентичности в SQL Server, исполняющая среда EF Core будет трактовать экземпляр сущности как добавленный (Added), если значение свойства первичного ключа равно 0. Следующий тест создает новую сущность Car и оставляет для свойства Id стандартное значение 0. После присоединения сущности к ChangeTracker ее состояние устанавливается в Added и вызов SaveChanges() добавит сущность в базу данных:

```

[Fact]
public void ShouldAddACarWithAttach()
{
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
        var car = new Car
        {
            Color = "Yellow",
            MakeId = 1,
            PetName = "Herbie"
        };
        var carCount = Context.Cars.Count();
        Context.Cars.Attach(car);
        Assert.Equal(EntityState.Added, Context.Entry(car).State);
        Context.SaveChanges();
        var newCarCount = Context.Cars.Count();
        Assert.Equal(carCount + 1, newCarCount);
    }
}

```

Добавление нескольких записей одновременно

Чтобы вставить в одной транзакции сразу несколько записей, применяйте метод AddRange() класса DbSet<T>, как показано в приведенном далее тесте (обратите внимание, что для активизации пакетирования при сохранении данных в SQL Server должно быть инициировано не менее четырех действий):

```
[Fact]
public void ShouldAddMultipleCars()
{
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
        // Для активизации пакетирования должны быть добавлены
        // четыре сущности.
        var cars = new List<Car>
        {
            new() { Color = "Yellow", MakeId = 1, PetName = "Herbie" },
            new() { Color = "White", MakeId = 2, PetName = "Mach 5" },
            new() { Color = "Pink", MakeId = 3, PetName = "Avon" },
            new() { Color = "Blue", MakeId = 4, PetName = "Blueberry" },
        };
        var carCount = Context.Cars.Count();
        Context.Cars.AddRange(cars);
        Context.SaveChanges();
        var newCarCount = Context.Cars.Count();
        Assert.Equal(carCount + 4, newCarCount);
    }
}
```

Операторы добавления пакетируются в единственное обращение к базе данных и запрашиваются все сгенерированные столбцы. Когда результаты запроса поступают в EF Core, сущности обновляются с использованием значений серверной стороны. Вот как выглядит выполняемый оператор SQL:

```
exec sp_executesql N'SET NOCOUNT ON;
DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
MERGE [dbo].[Inventory] USING (
VALUES (@p0, @p1, @p2, 0),
(@p3, @p4, @p5, 1),
(@p6, @p7, @p8, 2),
(@p9, @p10, @p11, 3)) AS i ([Color], [MakeId], [PetName], _Position) ON 1=0
WHEN NOT MATCHED THEN
INSERT ([Color], [MakeId], [PetName])
VALUES (i.[Color], i.[MakeId], i.[PetName])
OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;

SELECT [t].[Id], [t].[IsDriveable], [t].[TimeStamp]
FROM [dbo].[Inventory] t
INNER JOIN @inserted0 i ON ([t].[Id] = [i].[Id])
ORDER BY [i].[_Position];
N'@p0 nvarchar(50),@p1 int,@p2 nvarchar(50),@p3 nvarchar(50),
@p4 int,@p5 nvarchar(50),@p6 nvarchar(50),@p7 int,@p8 nvarchar(50),
```

```
@p9 nvarchar(50),@p10 int,@p11 nvarchar(50)',@p0=N'Yellow',@p1=1,
@p2=N'Herbie',@p3=N'White',@p4=2,@p5=N'Mach 5',@p6=N'Pink',@p7=3,
@p8=N'Avon',@p9=N'Blue',@p10=4,@p11=N'Blueberry'
```

Соображения относительно столбца идентичности при добавлении записей

Когда сущность имеет числовое свойство, которое определено как первичный ключ, то такое свойство (по умолчанию) отображается на столбец идентичности (`Identity`) в SQL Server. Исполняющая среда EF Core считает сущность со стандартным (нулевым) значением для свойства ключа новой, а сущность с нестандартным значением — уже присутствующей в базе данных. Если вы создаете новую сущность и устанавливаете свойство первичного ключа в ненулевое число, после чего пытаетесь добавить ее в базу данных, то EF Core откажется добавлять запись, поскольку вставка идентичности не разрешена. Включение вставки идентичности демонстрируется в коде инициализации данных.

Добавление объектного графа

При добавлении сущности в базу данных дочерние записи могут быть добавлены в том же самом обращении без их специального добавления в собственный экземпляр `DbSet<T>`, если они добавлены в свойство типа коллекции для родительской записи. Например, пусть создается новая сущность `Make` и в ее свойство `Cars` добавляется дочерняя запись `Car`. Когда сущность `Make` добавляется в свойство `DbSet<Make>`, исполняющая среда EF Core автоматически начинает отслеживание также и дочерней записи `Car` без необходимости в ее явном добавлении в свойство `DbSet<Car>`. Выполнение метода `SaveChanges()` приводит к совместному сохранению `Make` и `Car`, что демонстрируется в следующем тесте:

```
[Fact]
public void ShouldAddAnObjectGraph()
{
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
        var make = new Make {Name = "Honda"};
        var car = new Car {Color = "Yellow", MakeId = 1, PetName = "Herbie"};
        // Привести свойство Cars к List<Car> из IEnumerable<Car>.
        ((List<Car>)make.Cars).Add(car);
        Context.Makes.Add(make);
        var carCount = Context.Cars.Count();
        var makeCount = Context.Makes.Count();
        Context.SaveChanges();
        var newCarCount = Context.Cars.Count();
        var newMakeCount = Context.Makes.Count();
        Assert.Equal(carCount+1,newCarCount);
        Assert.Equal(makeCount+1,newMakeCount);
    }
}
```

Операторы добавления не пакетируются из-за наличия менее двух операторов, а в SQL Server пакетирование начинается с четырех операторов. Ниже показаны выполняемые операторы SQL:

```

exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [dbo].[Makes] ([Name])
VALUES (@p0);
SELECT [Id], [TimeStamp]
FROM [dbo].[Makes]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
',N'@p0 nvarchar(50)',@p0=N'Honda'
exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [dbo].[Inventory] ([Color], [MakeId], [PetName])
VALUES (@p1, @p2, @p3);
SELECT [Id], [IsDriveable], [TimeStamp]
FROM [dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
',N'@p1 nvarchar(50),@p2 int,@p3 nvarchar(50)',@p1=N'Yellow',@p2=7,
@p3=N'Herbie'

```

Обновление записей

Записи обновляются за счет их загрузки в `DbSet<T>` как отслеживаемой сущности, их изменения посредством кода и вызова метода `SaveChanges()` контекста. При выполнении `SaveChanges()` объект `ChangeTracker` сообщает обо всех модифицированных сущностях и исполняющая среда EF Core (наряду с поставщиком баз данных) создает надлежащий оператор SQL для обновления записи (или операторы SQL, если записей несколько).

Состояние сущности

Когда сущность редактируется, `EntityState` устанавливается в `Modified`. После успешного сохранения изменений состояние возвращается к `Unchanged`.

Обновление отслеживаемых сущностей

Обновление одиночной записи очень похоже на добавление одной записи. Вам понадобится загрузить запись из базы данных, внести в нее какие-то изменения и вызвать метод `SaveChanges()`. Обратите внимание, что вам не нужно вызывать `Update() / UpdateRange()` на экземпляре `DbSet<T>`, поскольку сущности отслеживаются. Представленный ниже тест обновляет только одну запись, но при обновлении и сохранении множества отслеживаемых сущностей процесс будет таким же:

```

[Fact]
public void ShouldUpdateACar()
{
    ExecuteInASharedTransaction(RunTheTest);
    void RunTheTest(IDbContextTransaction trans)
    {
        var car = Context.Cars.First(c => c.Id == 1);
        Assert.Equal("Black", car.Color);
        car.Color = "White";
        // Вызывать Update() не нужно, т.к. сущность отслеживается.
        // Context.Cars.Update(car);
        Context.SaveChanges();
        Assert.Equal("White", car.Color);
    }
}

```

```

    var context2 = TestHelpers.GetSecondContext(Context, trans);
    var car2 = context2.Cars.First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
}
}

```

В предыдущем коде задействована транзакция, совместно используемая двумя экземплярами `DbContext`. Это должно обеспечить изоляцию между контекстом, выполняющим тест, и контекстом, проверяющим результат теста.

Вот выполняемый оператор SQL:

```

exec sp_executesql N'SET NOCOUNT ON;
UPDATE [dbo].[Inventory] SET [Color] = @p0
WHERE [Id] = @p1 AND [TimeStamp] = @p2;
SELECT [TimeStamp]
FROM [dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = @p1;
',N'@p1 int,@p0 nvarchar(50),@p2 varbinary(8)',@p1=1,@p0=N'White',
@p2=0x0000000000000862D

```

На заметку! В показанной выше конструкции `WHERE` проверяется не только столбец `Id`, но и столбец `TimeStamp`. Проверка параллелизма будет раскрыта очень скоро.

Обновление неотслеживаемых сущностей

Неотслеживаемые сущности тоже можно использовать для обновления записей базы данных. Процесс аналогичен обновлению отслеживаемых сущностей за исключением того, что сущность создается в коде (и не запрашивается), а исполняющую среду EF Core потребуется уведомить о том, что сущность уже должна существовать в базе данных и нуждается в обновлении.

После создания экземпляра сущности есть два способа уведомления EF Core о том, что эту сущность необходимо обработать как обновление. Первый способ предусматривает вызов метода `Update()` на экземпляре `DbSet<T>`, который устанавливает состояние в `Modified`:

```
context2.Cars.Update(updatedCar);
```

Второй способ связан с применением экземпляра контекста и метода `Entry()` для установки состояния в `Modified`:

```
context2.Entry(updatedCar).State = EntityState.Modified;
```

В любом случае для сохранения значений все равно должен вызываться метод `SaveChanges()`.

В представленном далее тесте читается неотслеживаемая запись, из нее создается новый экземпляр класса `Car` и изменяется одно его свойство (`Color`). Затем в зависимости от того, с какой строки кода вы уберете комментарий, либо устанавливается состояние, либо использует метод `Update()` на `DbSet<T>`. Метод `Update()` также изменяет состояние на `Modified`. Затем в тесте вызывается метод `SaveChanges()`. Все дополнительные контексты нужны для обеспечения точности теста и отсутствия пересечения между контекстами:

```

[Fact]
public void ShouldUpdateACarUsingState()
{

```

```

ExecuteInASharedTransaction(RunTheTest);
void RunTheTest(DbContextTransaction trans)
{
    var car = Context.Cars.AsNoTracking().First(c => c.Id == 1);
    Assert.Equal("Black", car.Color);
    var updatedCar = new Car
    {
        Color = "White", // Первоначально был Black.
        Id = car.Id,
        MakeId = car.MakeId,
        PetName = car.PetName,
        TimeStamp = car.TimeStamp
        IsDriveable = car.IsDriveable
    };
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    // Либо вызвать Update(), либо модифицировать состояние.
    context2.Entry(updatedCar).State = EntityState.Modified;
    // context2.Cars.Update(updatedCar);
    context2.SaveChanges();
    var context3 =
        TestHelpers.GetSecondContext(Context, trans);
    var car2 = context3.Cars.First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
}
}

```

Ниже показан выполняющийся оператор SQL:

```

exec sp_executesql N'SET NOCOUNT ON;
UPDATE [dbo].[Inventory] SET [Color] = @p0
WHERE [Id] = @p1 AND [TimeStamp] = @p2;
SELECT [TimeStamp]
FROM [dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = @p1;
',N'@p1 int,@p0 nvarchar(50),@p2 varbinary(8)',@p1=1,@p0=N'White',
@p2=0x0000000000000862D

```

Проверка параллелизма

Проверка параллелизма подробно обсуждалась в предыдущей главе. Вспомните, что когда внутри сущности определено свойство `TimeStamp`, то значение этого свойства используется в конструкции `WHERE` при сохранении изменений (обновлений или удалений) в базе данных. Вместо поиска только первичного ключа к запросу добавляется поиск значения `TimeStamp`, например:

```

UPDATE [dbo].[Inventory] SET [PetName] = @p0
WHERE [Id] = @p1 AND [TimeStamp] = @p2;

```

В следующем тесте демонстрируется пример создания исключения, связанного с параллелизмом, его перехвата и применения `Entries` для получения исходных значений, текущих значений и значений, которые в настоящий момент хранятся в базе данных. Получение текущих значений требует еще одного обращения к базе данных:

```

[Fact]
public void ShouldThrowConcurrencyException()
{

```

```

ExecuteInATransaction(RunTheTest);
void RunTheTest()
{
    var car = Context.Cars.First();
    // Обновить базу данных за пределами контекста.
    Context.Database.ExecuteSqlInterpolated(
        $"Update dbo.Inventory set Color='Pink' where Id = {car.Id}");
    car.Color = "Yellow";
    var ex = Assert.Throws<CustomConcurrencyException>(
        () => Context.SaveChanges());
    var entry =
        ((DbUpdateConcurrencyException) ex.InnerException)?.Entries[0];
    PropertyValues originalProps = entry.OriginalValues;
    PropertyValues currentProps = entry.CurrentValues;
    // Требует еще одного обращения к базе данных.
    PropertyValues databaseProps = entry.GetDatabaseValues();
}
}

```

Ниже показаны выполняемые операторы SQL. Первым из них является оператор UPDATE, а вторым — обращение для получения значений базы данных:

```

exec sp_executesql N'SET NOCOUNT ON;
UPDATE [dbo].[Inventory] SET [Color] = @p0
WHERE [Id] = @p1 AND [TimeStamp] = @p2;
SELECT [TimeStamp]
FROM [dbo].[Inventory]
WHERE @@ROWCOUNT = 1 AND [Id] = @p1;
,N'@p1 int,@p0 nvarchar(50),@p2 varbinary(8)',@p1=1,@p0=N'Yellow',
@p2=0x00000000000008665
exec sp_executesql N'SELECT TOP(1) [i].[Id], [i].[Color],
    [i].[IsDriveable], [i].[MakeId], [i].[PetName], [i].[TimeStamp]
FROM [dbo].[Inventory] AS [i]
WHERE [i].[Id] = @_p_0',N' @_p_0 int', @_p_0=1

```

Удаление записей

Одиночная сущность помечается для удаления путем вызова `Remove()` на `DbSet<T>` или установки ее состояния в `Deleted`. Список записей помечается для удаления вызовом `RemoveRange()` на `DbSet<T>`. Процесс удаления будет вызывать эффекты каскадирования для навигационных свойств на основе правил, сконфигурированных в методе `OnModelCreating()` (и регламентированных соглашениями EF Core). Если удаление не допускается из-за политики каскадирования, тогда генерируется исключение.

Состояние сущности

Когда метод `Remove()` вызывается на отслеживаемой сущности, свойство `EntityState` устанавливается в `Deleted`. После успешного выполнения оператора удаления сущность исключается из `ChangeTracker` и состояние изменяется на `Detached`. Обратите внимание, что сущность по-прежнему существует в вашем приложении, если только она не покинула область видимости и не была подвержена сборке мусора.

Удаление отслеживаемых сущностей

Процесс удаления зеркально отображает процесс обновления. Как только сущность начала отслеживаться, вызовите `Remove()` на контексте и затем вызовите `SaveChanges()`, чтобы удалить запись из базы данных:

```
[Fact]
public void ShouldRemoveACar()
{
    ExecuteInATransaction(RunTheTest);

    void RunTheTest()
    {
        var carCount = Context.Cars.Count();
        var car = Context.Cars.First(c => c.Id == 2);
        Context.Cars.Remove(car);
        Context.SaveChanges();
        var newCarCount = Context.Cars.Count();
        Assert.Equal(carCount - 1, newCarCount);
        Assert.Equal(
            EntityState.Detached,
            Context.Entry(car).State);
    }
}
```

После вызова `SaveChanges()` экземпляр сущности все еще существует, но больше не находится в `ChangeTracker`. Состоянием `EntityState` будет `Detached`.

Вот как выглядит выполняемый код SQL:

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [dbo].[Inventory]
WHERE [Id] = @p0 AND [TimeStamp] = @p1;
SELECT @@ROWCOUNT;',
N'@p0 int,@p1 varbinary(8)',@p0=2,@p1=0x0000000000008680
```

Удаление неотслеживаемых сущностей

Неотслеживаемые сущности способны удалять записи таким же способом, каким они могут обновлять записи. Удаление производится вызовом `Remove()` / `RemoveRange()` или установкой состояния в `Deleted` и последующим вызовом `SaveChanges()`.

В показанном ниже тесте сначала читается запись как неотслеживаемая и на основе записи создается новый экземпляр класса `Car`. Затем либо устанавливается состояние в `Deleted`, либо применяется метод `Remove()` класса `DbSet<T>` (в зависимости от того, какая строка кода закомментирована) и вызывается `SaveChanges()`. Все дополнительные контексты нужны для обеспечения точности теста и отсутствия пересечения между контекстами:

```
[Fact]
public void ShouldRemoveACarUsingState()
{
    ExecuteInASharedTransaction(RunTheTest);

    void RunTheTest(IDbContextTransaction trans)
    {
        var carCount = Context.Cars.Count();
```

```

var car = Context.Cars.AsNoTracking().First(c => c.Id == 2);
var context2 = TestHelpers.GetSecondContext(Context, trans);

// Либо модифицировать состояние, либо вызвать Remove().
context2.Entry(car).State = EntityState.Deleted;

// context2.Cars.Remove(car);
context2.SaveChanges();
var newCarCount = Context.Cars.Count();
Assert.Equal(carCount - 1, newCarCount);
Assert.Equal(
    EntityState.Detached,
    Context.Entry(car).State);
}
}

```

Перехват отказов каскадного удаления

Когда попытка удаления записи терпит неудачу из-за правил каскадирования, то исполняющая среда EF Core генерирует исключение `DbUpdateException`. Следующий тест демонстрирует это в действии:

```

[Fact]
public void ShouldFailToRemoveACar()
{
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
        var car = Context.Cars.First(c => c.Id == 1);
        Context.Cars.Remove(car);
        Assert.Throws<CustomDbUpdateException>(
            ()=>Context.SaveChanges());
    }
}

```

Проверка параллелизма

Если сущность имеет свойство `TimeStamp`, то при удалении также используется проверка параллелизма. Дополнительную информацию ищите в подразделе “Проверка параллелизма” внутри раздела “Обновление записей” ранее в главе.

Резюме

В настоящей главе было закончено построение уровня доступа к данным AutoLot на основе сведений, полученных в предыдущей главе. С помощью инструментов командной строки EF Core вы создали шаблоны сущностей для существующей базы данных, обновили модель до финальной версии, а также создали и применили миграции. Для инкапсуляции доступа к данным вы добавили хранилища. Написанный код инициализации базы данных способен удалять и заново создавать базу данных повторяемым и надежным способом. В заключение готовый уровень доступа к данным главе был протестирован. На этом тема доступа к данным и Entity Framework Core завершена.

ЧАСТЬ VIII

**Разработка
клиентских приложений
для Windows**

ГЛАВА 24

Введение в Windows Presentation Foundation и XAML

Когда была выпущена версия 1.0 платформы .NET, программисты, нуждающиеся в построении графических настольных приложений, использовали два API-интерфейса под названиями Windows Forms и GDI+, упакованные преимущественно в сборках System.Windows.Forms.dll и System.Drawing.dll. Наряду с тем, что Windows Forms и GDI+ все еще являются жизнеспособными API-интерфейсами для построения традиционных настольных графических пользовательских интерфейсов, начиная с версии .NET 3.0, поставляется альтернативный API-интерфейс с таким же предназначением — Windows Presentation Foundation (WPF). В выпуске .NET Core 3.0 интерфейсы WPF и Windows Forms объединены с семейством .NET Core.

В начале этой вводной главы, посвященной WPF, вы ознакомитесь с мотивацией, лежащей в основе новой инфраструктуры для построения графических пользовательских интерфейсов, что поможет увидеть различия между моделями программирования Windows Forms/GDI+ и WPF. Затем анализируется роль ряда важных классов, включая Application, Window, ContentControl, Control, UIElement и FrameworkElement.

В настоящей главе будет представлена грамматика на основе XML, которая называется *расширяемым языком разметки приложений* (Extensible Application Markup Language — XAML). Вы изучите синтаксис и семантику XAML (в том числе синтаксис присоединяемых свойств, роль преобразователей типов и расширений разметки).

Глава завершается исследованием визуальных конструкторов WPF, встроенных в Visual Studio, за счет построения вашего первого приложения WPF. Вы научитесь перехватывать действия клавиатуры и мыши, определять данные уровня приложения и выполнять другие распространенные задачи WPF.

Побудительные причины создания WPF

На протяжении многих лет в Microsoft создавали инструменты для построения графических пользовательских интерфейсов (для низкоуровневой разработки на C/C++/Windows API, VB6, MFC и т.д.) настольных приложений. Каждый инструмент предлагает кодовую базу для представления основных аспектов приложения с графическим пользовательским интерфейсом, включая главные окна, диалоговые окна, элементы

управления, системы меню и другие базовые аспекты. После начального выпуска платформы .NET инфраструктура Windows Forms быстро стала предпочтительным подходом к разработке пользовательских интерфейсов благодаря своей простой, но очень мощной объектной модели.

Хотя с помощью Windows Forms было успешно создано множество полноценных настольных приложений, дело в том, что данная программная модель несколько ассиметрична. Попросту говоря, сборки `System.Windows.Forms.dll` и `System.Drawing.dll` не предоставляют прямую поддержку для многих дополнительных технологий, требуемых при построении полнофункционального настольного приложения. Чтобы проиллюстрировать сказанное, рассмотрим узкоспециализированную разработку графических пользовательских интерфейсов до выпуска WPF (табл. 24.1).

Таблица 24.1. Решения, предшествующие WPF, для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение окон с элементами управления	Windows Forms
Поддержка двумерной графики	GDI+ (<code>System.Drawing.dll</code>)
Поддержка трехмерной графики	API-интерфейсы DirectX
Поддержка потокового видео	API-интерфейсы Windows Media Player
Поддержка документов нефиксированного формата	Программное манипулирование файлами PDF

Как видите, разработчик, использующий Windows Forms, вынужден заимствовать типы из нескольких несвязанных API-интерфейсов и объектных моделей. Несмотря на то что применение всех разрозненных API-интерфейсов синтаксически похоже (в конце концов, это просто код C#), каждая технология требует радикально иного мышления. Например, навыки, необходимые для создания трехмерной анимации с использованием DirectX, совершенно отличаются от тех, что нужны для привязки данных к экранной сетке. Конечно, программисту Windows Forms чрезвычайно трудно в равной степени хорошо овладеть природой каждого API-интерфейса.

Унификация несходных API-интерфейсов

Инфраструктура WPF специально создавалась для объединения ранее несвязанных задач программирования в одну унифицированную объектную модель. Таким образом, при разработке трехмерной анимации больше не возникает необходимости в ручном кодировании с применением API-интерфейса DirectX (хотя это можно делать), поскольку нужная функциональность уже встроена в WPF. Чтобы продемонстрировать, насколько все стало яснее, в табл. 24.2 представлена модель разработки настольных приложений,веденная в .NET 3.0.

Очевидное преимущество здесь в том, что программисты приложений .NET теперь имеют единственный *симметричный API-интерфейс* для всех распространенных потребностей, появляющихся во время разработки графических пользовательских интерфейсов настольных приложений. Освоившись с функциональностью основных сборок WPF и грамматикой XAML, вы будете приятно удивлены, насколько быстро с их помощью можно создавать сложные пользовательские интерфейсы.

Таблица 24.2. Решения .NET 3.0 для обеспечения желаемой функциональности

Желаемая функциональность	Технология
Построение форм с элементами управления	WPF
Поддержка двумерной графики	WPF
Поддержка трехмерной графики	WPF
Поддержка потокового видео	WPF
Поддержка документов нефиксированного формата	WPF

Обеспечение разделения обязанностей через XAML

Возможно, одно из наиболее значительных преимуществ заключается в том, что инфраструктура WPF предлагает способ аккуратного отделения внешнего вида и поведения приложения с графическим пользовательским интерфейсом от программной логики, которая им управляет. Используя язык XAML, пользовательский интерфейс приложения можно определять через разметку XML. Такая разметка (в идеале генерируемая с помощью инструментов вроде Microsoft Visual Studio или Blend для Visual Studio) затем может быть подключена к связанному файлу кода для обеспечения внутренней части функциональности программы.

На заметку! Применение языка XAML не ограничено приложениями WPF. Любое приложение может использовать XAML для описания дерева объектов .NET, даже если они не имеют никакого отношения к видимому пользовательскому интерфейсу.

По мере погружения в WPF вас может удивить, насколько высокую гибкость обеспечивает эта "настольная разметка". Язык XAML позволяет определять в разметке не только простые элементы пользовательского интерфейса (кнопки, таблицы, окна со списками и т.д.), но также интерактивную двумерную и трехмерную графику, анимацию, логику привязки данных и функциональность мультимедиа (наподобие воспроизведения видео).

Кроме того, XAML облегчает настройку визуализации элемента управления. Например, определение круглой кнопки, на которой выполняется анимация логотипа компании, требует всего нескольких строк разметки. Как показано в главе 27, элементы управления WPF могут быть модифицированы посредством стилей и шаблонов, которые позволяют изменять весь внешний вид приложения с минимальными усилиями. В отличие от разработки с помощью Windows Forms единственной веской причиной для построения специального элемента управления WPF с нуля является необходимость в изменении поведения элемента управления (например, добавление специальных методов, свойств или событий либо создание подкласса существующего элемента управления с целью переопределения виртуальных членов). Если нужно просто изменить внешний вид элемента управления (как в случае с круглой анимированной кнопкой), то это можно делать полностью через разметку.

Обеспечение оптимизированной модели визуализации

Наборы инструментов для построения графических пользовательских интерфейсов, такие как Windows Forms, MFC или VB6, выполняют все запросы графической

визуализации (включая визуализацию элементов управления вроде кнопок и окон со списком) с применением низкоуровневого API-интерфейса на основе C (GDI), который в течение многих лет был частью Windows. Интерфейс GDI обеспечивает адекватную производительность для типовых бизнес-приложений или простых графических программ; однако если приложению с пользовательским интерфейсом нужна была высокопроизводительная графика, то приходилось обращаться к услугам DirectX.

Программная модель WPF полностью отличается тем, что при визуализации графических данных GDI не используется. Все операции визуализации (двумерная и трехмерная графика, анимация, визуализация элементов управления и т.д.) теперь работают с API-интерфейсом DirectX. Очевидная выгода такого подхода в том, что приложения WPF будут автоматически получать преимущества аппаратной и программной оптимизации. Вдобавок приложения WPF могут задействовать развитые графические службы (эффекты размытия, склаживания, прозрачности и т.п.) без сложностей, присущих программированию напрямую с применением API-интерфейса DirectX.

На заметку! Хотя WPF переносит все запросы визуализации на уровень DirectX, нельзя утверждать, что приложение WPF будет работать настолько же быстро, как приложение, построенное с использованием неуправляемого языка C++ и DirectX. Несмотря на значительные усовершенствования, вносимые в WPF с каждым новым выпуском, если вы намереваетесь строить настольное приложение, которое требует максимально возможной скорости выполнения (вроде трехмерной игры), то неуправляемый C++ и DirectX по-прежнему будут наилучшим выбором.

Упрощение программирования сложных пользовательских интерфейсов

Чтобы подвести итоги сказанному до сих пор: WPF — это API-интерфейс, предназначенный для построения настольных приложений, который интегрирует разнообразные настольные API-интерфейсы в единую объектную модель и обеспечивает четкое разделение обязанностей через XAML. В дополнение к указанным важнейшим моментам приложения WPF также выигрывают от простого способа интеграции со службами, что исторически было довольно сложным. Ниже кратко перечислены основные функциональные возможности WPF.

- Множество диспетчеров компоновки (гораздо больше, чем в Windows Forms) для обеспечения исключительно гибкого контроля над размещением и изменением позиций содержимого.
- Применение расширенного механизма привязки данных для связывания содержимого с элементами пользовательского интерфейса разнообразными способами.
- Встроенный механизм стилей, который позволяет определять “темы” для приложения WPF.
- Использование векторной графики, поддерживающей автоматическое изменение размеров содержимого с целью соответствия размерам и разрешающей способности экрана, который отображает пользовательский интерфейс приложения.
- Поддержка двумерной и трехмерной графики, анимации, а также воспроизведения видео- и аудио-роликов.

- Развитый типографский API-интерфейс, который поддерживает документы XML Paper Specification (XPS), фиксированные документы (WYSIWYG), документы нефиксированного формата и аннотации в документах (например, API-интерфейс Sticky Notes).
- Поддержка взаимодействия с унаследованными моделями графических пользовательских интерфейсов (такими как Windows Forms, ActiveX и HWND-дескрипторы Win32). Например, в приложение WPF можно встраивать специальные элементы управления Windows Forms и наоборот.

Теперь, получив определенное представление о том, что инфраструктура WPF приносит в платформу, давайте рассмотрим разнообразные типы приложений, которые могут быть созданы с применением данного API-интерфейса. Многие из перечисленных выше возможностей будут подробно исследованы в последующих главах.

Исследование сборок WPF

В конечном итоге инфраструктура WPF — не многим более чем коллекция типов, встроенных в сборки .NET Core. В табл. 24.3 описаны основные сборки, используемые при разработке приложений WPF, на каждую из которых должна быть добавлена ссылка, когда создается новый проект. Как и следовало ожидать, проекты WPF в Visual Studioзываются на эти обязательные сборки автоматически.

Таблица 24.3. Основные сборки WPF

Сборка	Описание
PresentationCore.dll	В этой сборке определены многочисленные пространства имен, которые образуют фундамент уровня графического пользовательского интерфейса в WPF. Например, она включает поддержку API-интерфейса WPF Ink, примитивы анимации и множество типов графической визуализации
PresentationFramework.dll	В этой сборке содержится большинство элементов управления WPF, классы Application и Window, поддержка интерактивной двумерной графики и многочисленные типы, применяемые для привязки данных
System.Xaml.dll	Эта сборка предоставляет пространства имен, которые позволяют программно взаимодействовать с документами XAML во время выполнения. В общем и целом она полезна только при разработке инструментов поддержки WPF или когда нужен абсолютный контроль над разметкой XAML во время выполнения
WindowsBase.dll	В этой сборке определены типы, которые формируют инфраструктуру API-интерфейса WPF, включая типы потоков WPF, типы, связанные с безопасностью, разнообразные преобразователи типов и поддержку свойств зависимости и маршрутизируемых событий (описанных в главе 25)

В этих четырех сборках определены новые пространства имен, а также классы, интерфейсы, структуры, перечисления и делегаты .NET Core. В табл. 24.4 описана роль некоторых (но далеко не всех) важных пространств имен.

Таблица 24.4. Основные пространства имен WPF

Пространство имен	Описание
System.Windows	Корневое пространство имен WPF. Здесь находятся основные классы (такие как Application и Window), которые требуются в любом проекте настольного приложения WPF
System.Windows.Controls	Содержит все ожидаемые графические элементы (виджеты) WPF, включая типы для построения систем меню, всплывающие подсказки и многочисленные диспетчеры компоновки
System.Windows.Data	Содержит типы для работы с механизмом привязки данных WPF, а также для поддержки шаблонов привязки данных
System.Windows.Documents	Содержит типы для работы с API-интерфейсом документов, который позволяет интегрировать в приложения WPF функциональность в стиле PDF через протокол XML Paper Specification (XPS)
System.Windows.Ink	Предоставляет поддержку Ink API — интерфейса, который позволяет получать ввод от пера или мыши, реагировать на входные жесты и т.д. Этот API-интерфейс полезен при программировании для Tablet PC, но может использоваться также в любых приложениях WPF
System.Windows.Markup	Здесь определено множество типов, обеспечивающих программный анализ и обработку разметки XAML (и эквивалентного двоичного формата BAML)
System.Windows.Media	Корневое пространство имен для нескольких связанных с мультимедиа пространств имен, внутри которых определены типы для работы с анимацией, визуализацией трехмерной графики, визуализацией текста и прочими мультимедийными примитивами
System.Windows.Navigation	Предоставляет типы для обеспечения логики навигации, применяемой браузерными приложениями XAML (XAML browser application — XBAP), а также настольными приложениями, которые требуют страничной модели навигации
System.Windows.Shapes	Здесь определены классы, которые позволяют визуализировать двумерную графику, автоматически реагирующую на ввод с помощью мыши

В начале путешествия по программной модели WPF вы исследуете два члена пространства имен `System.Windows`, которые являются общими при традиционной разработке любого настольного приложения: `Application` и `Window`.

На заметку! Если вы создавали пользовательские интерфейсы для настольных приложений с использованием API-интерфейса Windows Forms, то имейте в виду, что сборки `System.Windows.Forms.*` и `System.Drawing.*` никак не связаны с WPF. Они относятся к первоначальному инструментальному набору .NET для построения графических пользовательских интерфейсов, т.е. Windows Forms/GDI+.

Роль класса Application

Класс `System.Windows.Application` представляет глобальный экземпляр выполняющегося приложения WPF. В нем имеется метод `Run()` (для запуска приложения) и комплект событий, которые можно обрабатывать для взаимодействия с приложением на протяжении его времени жизни (наподобие `Startup` и `Exit`). В табл. 24.5 описаны основные свойства класса `Application`.

Таблица 24.5. Основные свойства класса Application

Свойство	Описание
<code>Current</code>	Это статическое свойство позволяет получать доступ к работающему объекту <code>Application</code> из любого места кода. Может быть полезно, когда обычному или диалоговому окну необходим доступ к создавшему его объекту <code>Application</code> , как правило, для взаимодействия с переменными или функциональностью уровня приложения
<code>MainWindow</code>	Это свойство позволяет программно получать или устанавливать главное окно приложения
<code>Properties</code>	Это свойство позволяет устанавливать и получать данные, доступные через все аспекты приложения WPF (окна, диалоговые окна и т.д.)
<code>StartupUri</code>	Это свойство получает или устанавливает URI, который указывает окно или страницу для автоматического открытия при запуске приложения
<code>Windows</code>	Это свойство возвращает объект типа <code>WindowCollection</code> , предоставляющий доступ ко всем окнам, которые созданы в потоке, создавшем объект <code>Application</code> . Может быть удобным, когда нужно пройти по всем открытым окнам приложения и изменить их состояние (скажем, свернуть все окна)

Построение класса приложения

В любом приложении WPF нужно будет определить класс, расширяющий `Application`. Внутри такого класса определяется точка входа программы (метод `Main()`), которая создает экземпляр данного подкласса и обычно обрабатывает события `Startup` и `Exit` (при необходимости). Вот пример:

```
// Определить глобальный объект приложения для этой программы WPF.
class MyApp : Application
{
    [STAThread]
```

```

static void Main(string[] args)
{
    // Создать объект приложения.
    MyApp app = new MyApp();

    // Зарегистрировать события Startup/Exit.
    app.Startup += (s, e) => { /* Запуск приложения */ };
    app.Exit += (s, e) => { /* Завершение приложения */ };
}
}

```

В обработчике события Startup чаще всего обрабатываются входные аргументы командной строки и запускается главное окно программы. Как и следовало ожидать, обработчик события Exit представляет собой место, куда можно поместить любую необходимую логику завершения программы (например, сохранение пользовательских предпочтений).

На заметку! Метод Main() приложения WPF должен быть снабжен атрибутом [STAThread], который гарантирует, что любые унаследованные объекты COM, используемые приложением, являются безопасными в отношении потоков. Если не аннотировать метод Main() подобным образом, тогда во время выполнения возникнет исключение. Даже после появления в версии C# 9.0 операторов верхнего уровня вы все равно будете стремиться использовать в приложениях WPF традиционный метод Main(). В действительности метод Main() генерируется автоматически.

Перечисление элементов коллекции Windows

Еще одним интересным свойством класса Application является Windows, обеспечивающее доступ к коллекции, которая представляет все окна, загруженные в память для текущего приложения WPF. Вспомните, что создаваемые новые объекты Window автоматически добавляются в коллекцию Application.Windows. Ниже приведен пример метода, который сворачивает все окна приложения (возможно в ответ на нажатие определенного сочетания клавиш или выбор пункта меню конечным пользователем):

```

static void MinimizeAllWindows()
{
    foreach (Window wnd in Application.Current.Windows)
    {
        wnd.WindowState = WindowState.Minimized;
    }
}

```

Вскоре будет построено несколько приложений WPF, а пока давайте выясним основную функциональность типа Window и изучим несколько базовых классов WPF.

Роль класса Window

Класс System.Windows.Window (из сборки PresentationFramework.dll) представляет одиночное окно, которым владеет производный от Application класс, включая все отображаемые главным окном диалоговые окна. Тип Window вполне ожидаемо имеет несколько родительских классов, каждый из которых привносит дополнительную функциональность.

На рис. 24.1 показана цепочка наследования (и реализуемые интерфейсы) для класса `System.Windows.Window`, как она выглядит в браузере объектов Visual Studio.

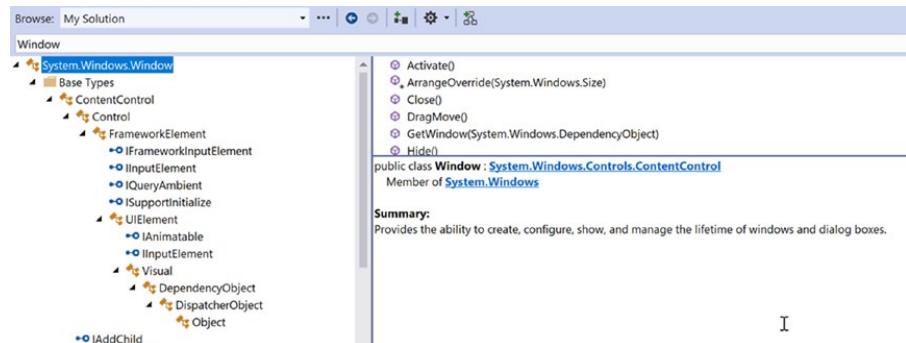


Рис. 24.1. Иерархия наследования класса Window

По мере чтения этой и последующих глав вы начнете понимать функциональность, предлагаемую многими базовыми классами WPF. Далее представлен краткий обзор функциональности каждого базового класса (полные сведения ищите в документации по .NET 5).

Роль класса `System.Windows.Controls.ContentControl`

Непосредственным родительским классом `Window` является класс `ContentControl`, который вполне можно считать самым впечатляющим из всех классов WPF. Базовый класс `ContentControl` снабжает производные типы способностью размещать в себе одиничный фрагмент содержимого, который, выражаясь упрощенно, относится к визуальным данным, помещенным внутрь области элемента управления через свойство `Content`. Модель содержимого WPF позволяет довольно легко настраивать базовый вид и поведение элемента управления `ContentControl`.

Например, когда речь идет о типичном “кнопочном” элементе управления, то обычно предполагается, что его содержимым будет простой строковый литерал (OK, Cancel, Abort и т.д.). Если для описания элемента управления WPF применяется XAML, а значение, которое необходимо присвоить свойству `Content`, может быть выражено в виде простой строки, тогда вот как установить свойство `Content` внутри открывшегося определения элемента:

```
<!-- Установка значения Content в открывающем элементе -->
<Button Height="80" Width="100" Content="OK"/>
```

На заметку! Свойство `Content` можно также устанавливать в коде C#, что позволяет изменять внутренности элемента управления во время выполнения.

Однако содержимое может быть практически любым. Например, пусть нужна “кнопка”, которая содержит в себе что-то более интересное, нежели простую строку — возможно специальную графику или текстовый фрагмент. В других инфраструктурах для построения пользовательских интерфейсов, таких как Windows Forms, потребовалось бы создать специальный элемент управления, что могло повлечь за собой

написание значительного объема кода и сопровождение полностью нового класса. Благодаря модели содержимого WPF необходимость в этом отпадает.

Когда свойству Content должно быть присвоено значение, которое невозможно выразить в виде простого массива символов, его нельзя присвоить с использованием атрибута в открывающем определении элемента управления. Взамен понадобится определить данные содержимого *неявно* внутри области действия элемента. Например, следующий элемент <Button> включает в качестве содержимого элемент <StackPanel>, который сам имеет уникальные данные (а именно — <Ellipse> и <Label>):

```
<!-- Неявная установка для свойства Content сложных данных -->
<Button Height="80" Width="100">
    <StackPanel>
        <Ellipse Fill="Red" Width="25" Height="25"/>
        <Label Content ="OK!" />
    </StackPanel>
</Button>
```

Для установки сложного содержимого можно также применять синтаксис “свойство-элемент” языка XAML. Взгляните на показанное далее функционально эквивалентное определение <Button>, которое явно устанавливает свойство Content с помощью синтаксиса “свойство-элемент” (дополнительная информация о XAML будет дана позже в главе, так что пока не обращайте внимания на детали):

```
<!-- Установка свойства Content с использованием
    синтаксиса "свойство-элемент" -->
<Button Height="80" Width="100">
    <Button.Content>
        <StackPanel>
            <Ellipse Fill="Red" Width="25" Height="25"/>
            <Label Content ="OK!" />
        </StackPanel>
    </Button.Content>
</Button>
```

Имейте в виду, что не каждый элемент WPF является производным от класса ContentControl, поэтому не все элементы поддерживают такую уникальную модель содержимого (хотя большинство поддерживает). Кроме того, некоторые элементы управления WPF вносят несколько усовершенствований в только что рассмотренную базовую модель содержимого. В главе 25 роль содержимого WPF раскрывается более подробно.

Роль класса System.Windows.Controls.Control

В отличие от ContentControl все элементы управления WPF разделяются в качестве общего родительского класса базовый класс Control. Он предоставляет многочисленные члены, которые необходимы для обеспечения основной функциональности пользовательского интерфейса. Например, в классе Control определены свойства для установки размеров элемента управления, прозрачности, порядка обхода по нажатию клавиши <Tab>, отображаемого курсора, цвета фона и т.д. Более того, данный родительский класс предлагает поддержку шаблонных служб. Как объясняется в главе 27, элементы управления WPF могут полностью изменять способ визуализации своего внешнего вида, используя шаблоны и стили. В табл. 24.6 кратко описаны основные члены типа Control, сгруппированные по связанный функциональности.

Таблица 24.6. Основные члены класса Control

Член	Описание
Background, Foreground, BorderBrush, BorderThickness, Padding, HorizontalContentAlignment, VerticalContentAlignment	Эти свойства позволяют устанавливать базовые настройки, касающиеся того, как элемент управления будет визуализироваться и позиционироваться
FontFamily, FontSize, FontStretch, FontWeight	Эти свойства управляют разнообразными настройками шрифтов
IstabStop, TabIndex	Эти свойства позволяют устанавливать порядок обхода элементов управления в окне по нажатию клавиши <Tab>
MouseDoubleClick, PreviewMouseDoubleClick	Эти события обрабатывают действие двойного щелчка на виджете
Template	Это свойство применяется для получения и установки шаблона элемента управления, который может быть использован для изменения вывода визуализации виджета

Роль класса System.Windows.FrameworkElement

Базовый класс FrameworkElement предоставляет несколько членов, которые применяются повсюду в инфраструктуре WPF, в том числе для поддержки раскадровки (в целях анимации) и привязки данных, а также возможности именования членов (через свойство Name), получения любых ресурсов, определенных производным типом, и установки общих измерений производного типа. Основные члены класса FrameworkElement кратко описаны в табл. 24.7.

Таблица 24.7. Основные члены класса FrameworkElement

Член	Описание
ActualHeight, ActualWidth, MaxHeight, MaxWidth, MinHeight, MinWidth, Height, Width	Эти свойства управляют размерами производного типа
ContextMenu	Это свойство получает или устанавливает всплывающее меню, ассоциированное с производным типом
Cursor	Это свойство получает или устанавливает курсор мыши, ассоциированный с производным типом
HorizontalAlignment, VerticalAlignment	Это свойство управляет позиционированием типа внутри контейнера (такого как панель или окно со списком)
Name	Это свойство позволяет назначать имя типу, чтобы обращаться к его функциональности в файле кода
Resources	Это свойство предоставляет доступ к любым ресурсам, которые определены типом (система ресурсов WPF объясняется в главе 27)
ToolTip	Это свойство получает или устанавливает всплывающую подсказку, ассоциированную с производным типом

Роль класса *System.Windows.UIElement*

Из всех типов в цепочке наследования класса Window наибольший объем функциональности обеспечивает базовый класс UIElement. Его основная задача — предоставить производному типу многочисленные события, чтобы он мог получать фокус и обрабатывать входные запросы. Например, в классе UIElement предусмотрены многочисленные события для обслуживания операций перетаскивания, перемещения курсора мыши, клавиатурного ввода, ввода посредством пера и сенсорного ввода.

Модель событий WPF будет подробно описана в главе 25; тем не менее, многие основные события будут выглядеть вполне знакомо (MouseMove, MouseDown, MouseEnter, MouseLeave, KeyUp и т.д.). В дополнение к десяткам событий родительский класс UIElement предлагает свойства, предназначенные для управления фокусом, состоянием доступности, видимостью и логикой проверки попадания (табл. 24.8).

Таблица 24.8. Основные члены класса *UIElement*

Член	Описание
Focusable, IsFocused	Эти свойства позволяют устанавливать фокус на заданный производный тип
.IsEnabled	Это свойство позволяет управлять доступностью заданного производного типа
IsMouseDirectlyOver, IsMouseOver	Эти свойства предоставляют простой способ выполнения логики проверки попадания
IsVisible, Visibility	Эти свойства позволяют работать с настройкой видимости производного типа
RenderTransform	Это свойство позволяет устанавливать трансформацию, которая будет использоваться при визуализации производного типа

Роль класса *System.Windows.Media.Visual*

Класс Visual предлагает основную поддержку визуализации в WPF, которая включает проверку попадания для графических данных, координатную трансформацию и вычисление ограничивающих прямоугольников. В действительности при рисовании данных на экране класс Visual взаимодействует с подсистемой DirectX. Как будет показано в главе 26, инфраструктура WPF поддерживает три возможных способа визуализации графических данных, каждый из которых отличается в плане функциональности и производительности. Применение типа Visual (и его потомков вроде DrawingVisual) является наиболее легковесным путем визуализации графических данных, но также подразумевает написание вручную большого объема кода для учета всех требуемых служб. Более подробно об этом пойдет речь в главе 26.

Роль класса *System.Windows.DependencyObject*

Инфраструктура WPF поддерживает отдельную разновидность свойств .NET под названием *свойства зависимости*. Выражаясь упрощенно, данный стиль свойств предоставляет дополнительный код, чтобы позволить свойству реагировать на определенные технологии WPF, такие как стили, привязка данных, анимация и т.д. Чтобы тип поддерживал подобную схему свойств, он должен быть производным от базового

класса `DependencyObject`. Несмотря на то что свойства зависимости являются ключевым аспектом разработки WPF, большую часть времени их детали скрыты от глаз. В главе 25 мы рассмотрим свойства зависимости более подробно.

Роль класса `System.Windows.Threading.DispatcherObject`

Последним базовым классом для типа `Window` (помимо `System.Object`, который здесь не требует дополнительных пояснений) является `DispatcherObject`. В нем определено одно интересное свойство `Dispatcher`, которое возвращает ассоциированный объект `System.Windows.Threading.Dispatcher`. Класс `Dispatcher` — это точка входа в очередь событий приложения WPF, и он предоставляет базовые конструкции для организации параллелизма и многопоточности. Объект `Dispatcher` обсуждался в главе 15.

Синтаксис XAML для WPF

Приложения WPF производственного уровня обычно будут использовать отдельные инструменты для генерации необходимой разметки XAML. Как бы ни были удобны такие инструменты, важно понимать общую структуру языка XAML. Для содействия процессу изучения доступен популярный (и бесплатный) инструмент, который позволяет легко экспериментировать с XAML.

Введение в Kaxaml

Когда вы только приступаете к изучению грамматики XAML, может оказаться удобным в применении бесплатный инструмент под названием `Kaxaml`. Этот популярный редактор/анализатор XAML доступен по ссылке <https://github.com/punker76/kaxaml>.

На заметку! Во многих предшествующих изданиях книги мы направляли читателей на веб-сайт www.kaxaml.com, но, к сожалению, он прекратил свою работу. Ян Каргер (<https://github.com/punker76>) сделал ответвление от старого кода и потрудился над его улучшением. Его версия инструмента доступна в GitHub по ссылке <https://github.com/punker76/kaxaml/releases>. Стоит выразить благодарность создателям за великолепный инструмент Kaxaml и Яну за то, что он сохранил его; Kaxaml помог многочисленным разработчикам изучить XAML.

Редактор Kaxaml полезен тем, что не имеет никакого понятия об исходном коде C#, обработчиках ошибок или логике реализации. Он предлагает намного более прямолинейный способ тестирования фрагментов XAML, нежели использование полноценного шаблона проекта WPF в Visual Studio. К тому же Kaxaml обладает набором интегрированных инструментов, в том числе средством выбора цвета, диспетчером фрагментов XAML и даже средством “очистки XAML”, которое форматирует разметку XAML на основе заданных настроек. Открыв Kaxaml в первый раз, вы найдете в нем простую разметку для элемента управления `<Page>`:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    </Grid>
</Page>
```

Подобно объекту *Window* объект *Page* содержит разнообразные диспетчеры компоновки и элементы управления. Тем не менее, в отличие от *Window* объекты *Page* не могут запускаться как отдельные сущности. Взамен они должны помещаться внутри подходящего хоста, такого как *NavigationWindow* или *Frame*. Хорошая новость в том, что в элементах *<Page>* и *<Window>* можно вводить идентичную разметку.

На заметку! Если в окне разметки Kaxaml заменить элементы *<Page>* и *</Page>* элементами *<Window>* и *</Window>*, тогда можно нажать клавишу *F5* и отобразить на экране новое окно.

В качестве начального теста введите следующую разметку в панели XAML, находящейся в нижней части окна Kaxaml:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <!-- Кнопка со специальным содержимым -->
        <Button Height="100" Width="100">
            <Ellipse Fill="Green" Height="50" Width="50"/>
        </Button>
    </Grid>
</Page>
```

В верхней части окна Kaxaml появится визуализированная страница (рис. 24.2).

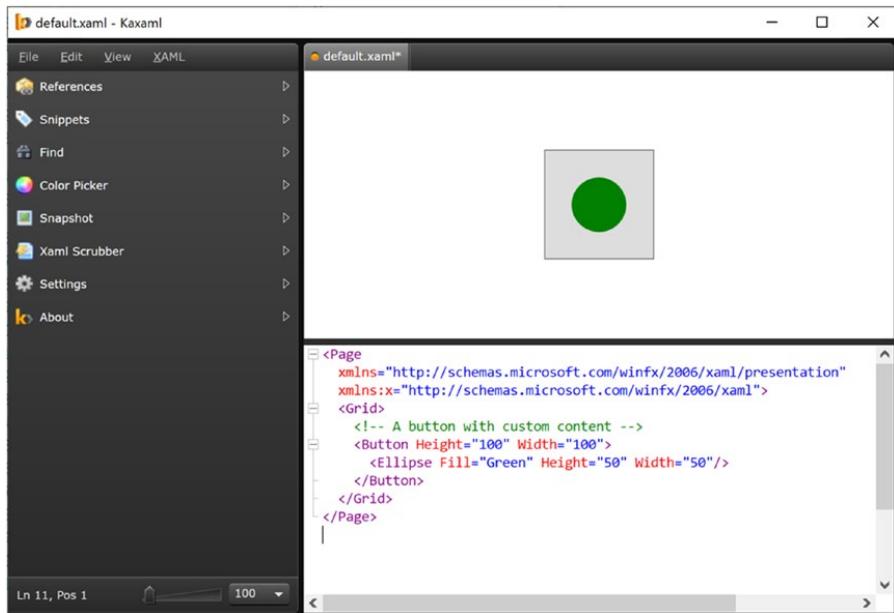


Рис. 24.2. Редактор Kaxaml является удобным (и бесплатным) инструментом, применяемым при изучении грамматики XAML

Во время работы с XAML помните, что данный инструмент не позволяет писать разметку, которая влечет за собой любую компиляцию кода (но разрешено использовать `x:Name`). Сюда входит определение атрибута `x:Class` (для указания файла кода), ввод имен обработчиков событий в разметке или применение любых ключевых слов XAML, которые также предусматривают компиляцию кода (вроде `FieldModifier` или `ClassModifier`). Попытка поступить так приводит к ошибке разметки.

Пространства имен XML и “ключевые слова” XAML

Корневой элемент XAML-документа WPF (такой как `<Window>`, `<Page>`, `<UserControl>` или `<Application>`) почти всегда будет ссылаться на два заранее определенные пространства имен XML:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
  </Grid>
</Page>
```

Первое пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, отображает множество связанных с WPF пространств имен .NET для использования текущим файлом `*.xaml` (`System.Windows`, `System.Windows.Controls`, `System.Windows.Data`, `System.Windows.Ink`, `System.Windows.Media`, `System.Windows.Navigation` и т.д.).

Это отображение “один ко многим” в действительности жестко закодировано внутри сборок WPF (`WindowsBase.dll`, `PresentationCore.dll` и `PresentationFramework.dll`) с применением атрибута `[XmlnsDefinition]` уровня сборки. Например, если открыть браузер объектов Visual Studio и выбрать сборку `PresentationCore.dll`, то можно увидеть списки, подобные показанному ниже, в котором импортируется пространство имен `System.Windows`:

```
[assembly: XmlnsDefinition(
    "http://schemas.microsoft.com/winfx/2006/xaml/presentation",
    "System.Windows")]
```

Второе пространство имен XML, `http://schemas.microsoft.com/winfx/2006/xaml`, используется для добавления специфичных для XAML “ключевых слов” (термин выбран за неимением лучшего), а также пространства имен `System.Windows.Markup`:

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
    "System.Windows.Markup")]
```

Одно из правил любого корректно сформированного документа XML (не забывайте, что грамматика XAML основана на XML) состоит в том, что открывающий корневой элемент назначает одно пространство имен XML в качестве *первичного пространства имен*, которое обычно представляет собой пространство имен, содержащее самые часто применяемые элементы. Если корневой элемент требует включения дополнительных вторичных пространств имен (как видно здесь), то они должны быть определены с использованием уникального префикса (чтобы устраниТЬ возможные конфликты имен). По соглашению для префикса применяется просто `x`, однако он может быть любым уникальным маркером, таким как `XamlSpecificStuff`:

```

<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <!-- Кнопка со специальным содержимым -->
        <Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
            <Ellipse Fill="Green" Height="50" Width="50"/>
        </Button>
    </Grid>
</Page>

```

Очевидный недостаток определения длинных префиксов для пространств имен XML связан с тем, что XamlSpecificStuff придется набирать всякий раз, когда в файле XAML нужно сослаться на один из элементов, определенных в этом пространстве имен XML. Из-за того, что префикс XamlSpecificStuff намного длиннее, давайте ограничимся x.

Помимо ключевых слов x:Name, x:Class и x:Code пространство имен http://schemas.microsoft.com/winfx/2006/xaml также предоставляет доступ к дополнительным ключевым словам XAML, наиболее распространенные из которых кратко описаны в табл. 24.9.

Таблица 24.9. Ключевые слова XAML

Ключевое слово XAML	Описание
x:Array	Представляет в XAML тип массива .NET
x:ClassModifier	Позволяет определять видимость класса C# (internal или public), обозначенного ключевым словом Class
x:FieldModifier	Позволяет определять видимость члена типа (internal, public, private или protected) для любого именованного подэлемента корня (например, <Button> внутри элемента <Window>). Именованный элемент определяется с использованием ключевого слова Name в XAML
x:Key	Позволяет устанавливать значение ключа для элемента XAML, которое будет помещено в элемент словаря
x:Name	Позволяет указывать сгенерированное имя C# заданного элемента XAML
x:Null	Представляет ссылку null
x:Static	Позволяет ссылаться на статический член типа
x>Type	Эквивалент XAML операции typeof языка C# (она будет выдавать объект System.Type на основе предоставленного имени)
x>TypeArguments	Позволяет устанавливать элемент как обобщенный тип с определенным параметром типа (например, List<int> или List<bool>)

В дополнение к двум указанным объявлениям пространств имен XML можно (а иногда и нужно) определить дополнительные префиксы дескрипторов в открывающем элементе документа XAML. Обычно так поступают, когда необходимо описать в XAML класс .NET Core, определенный во внешней сборке.

Например, предположим, что было построено несколько специальных элементов управления WPF, которые упакованы в библиотеку по имени MyControls.dll. Если теперь требуется создать новый объект Window, в котором применяются созданные элементы, то можно установить специальное пространство имен XML, отображаемое на библиотеку MyControls.dll, с использованием маркеров `clr-namespace` и `assembly`. Ниже приведен пример разметки, создающей префикс дескриптора по имени `myCtrls`, который может применяться для доступа к элементам управления в этой библиотеке:

```
<Window x:Class="WpfApplication1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
    Title="MainWindow" Height="350" Width="525">
    <Grid>
        <myCtrls:MyCustomControl />
    </Grid>
</Window>
```

Маркеру `clr-namespace` назначается название пространства имен .NET Core в сборке, в то время как маркер `assembly` устанавливается в дружественное имя внешней сборки *.dll. Такой синтаксис можно использовать для любой внешней библиотеки .NET Core, которой желательно манипулировать внутри разметки. В настоящее время в этом нет необходимости, но в последующих главах понадобится определять специальные объявления пространств имен XML для описания типов в разметке.

На заметку! Если нужно определить в разметке класс, который является частью текущей сборки, но находится в другом пространстве имен .NET Core, то префикс дескриптора `xmlns` определяется без атрибута `assembly`:

```
xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"
```

Управление видимостью классов и переменных-членов

Многие ключевые слова вы увидите в действии в последующих главах там, где они потребуются; тем не менее, в качестве простого примера взгляните на следующее XAML-определение `<Window>`, в котором применяются ключевые слова `ClassModifier` и `FieldModifier`, а также `x:Name` и `x:Class` (вспомните, что редактор Kaxaml не позволяет использовать ключевые слова XAML, вовлекающие компиляцию, такие как `x:Code`, `x:FieldModifier` или `x:ClassModifier`):

```
<!-- Этот класс теперь будет объявлен как internal в файле *.g.cs -->
<Window x:Class="MyWPFApp.MainWindow" x:ClassModifier ="internal"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- Эта кнопка будет объявлена как public в файле *.g.cs -->
    <Button x:Name ="myButton" x:FieldModifier ="public" Content = "OK"/>
</Window>
```

По умолчанию все определения типов C#/XAML являются открытыми (`public`), а члены — внутренними (`internal`). Однако для показанного выше определения XAML результирующий автоматически сгенерированный файл содержит внутренний тип класса с открытой переменной-членом `Button`:

```
internal partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    public System.Windows.Controls.Button myButton;
    ...
}
```

Элементы XAML, атрибуты XAML и преобразователи типов

После установки корневого элемента и необходимых пространств имен XML следующая задача заключается в наполнении корня *дочерним элементом*. В реальном приложении WPF дочерним элементом будет диспетчер компоновки (такой как Grid или StackPanel), который в свою очередь содержит любое количество дополнительных элементов, описывающих пользовательский интерфейс. Такие диспетчеры компоновки рассматриваются в главе 25, а пока предположим, что элемент <Window> будет содержать единственный элемент Button.

Как было показано ранее в главе, элементы XAML отображаются на типы классов или структур внутри заданного пространства имен .NET Core, тогда как атрибуты в открывающем дескрипторе элемента отображаются на свойства или события конкретного типа. В целях иллюстрации введите в редакторе XAML следующее определение <Button>:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <!-- Сконфигурировать внешний вид элемента Button -->
        <Button Height="50" Width="100" Content="OK!" 
            FontSize="20" Background="Green" Foreground="Yellow"/>
    </Grid>
</Page>
```

Обратите внимание, что присвоенные свойствам значения представлены с помощью простого текста. Это может выглядеть как полное несоответствие типам данных, поскольку после создания такого элемента Button в коде C# данным свойствам будут присваиваться *не* строковые объекты, а значения специфических типов данных. Например, ниже показано, как та же самая кнопка описана в коде:

```
public void MakeAButton()
{
    Button myBtn = new Button();
    myBtn.Height = 50;
    myBtn.Width = 100;
    myBtn.FontSize = 20;
    myBtn.Content = "OK!";
    myBtn.Background = new SolidColorBrush(Colors.Green);
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Оказывается, что инфраструктура WPF поставляется с несколькими классами *преобразователей типов*, которые будут применяться для трансформации простых текстовых значений в корректные типы данных. Такой процесс происходит прозрачно (и автоматически).

Тем не менее, нередко возникает потребность в присваивании атрибуту XAML на многое более сложного значения, которое невозможно выразить посредством простой строки. Например, пусть необходимо построить специальную кисть для установки свойства `Background` элемента `Button`. Создать кисть подобного рода в коде довольно просто:

```
public void MakeAButton()
{
    ...
    // Необычная кисть для фона.
    LinearGradientBrush fancyBruch =
        new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen, 45);
    myBtn.Background = fancyBruch;
    myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Но можно ли представить эту сложную кисть в виде строки? Нет, нельзя! К счастью, в XAML предусмотрен специальный синтаксис, который можно использовать всякий раз, когда нужно присвоить сложный объект в качестве значения свойства; он называется *синтаксисом “свойство-элемент”*.

Понятие синтаксиса “свойство-элемент” в XAML

Синтаксис “свойство-элемент” позволяет присваивать свойству сложные объекты. Ниже показано описание XAML элемента `Button`, в котором для установки свойства `Background` применяется объект `LinearGradientBrush`:

```
<Button Height="50" Width="100" Content="OK!"  
       FontSize="20" Foreground="Yellow">  
    <Button.Background>  
        <LinearGradientBrush>  
            <GradientStop Color="DarkGreen" Offset="0"/>  
            <GradientStop Color="LightGreen" Offset="1"/>  
        </LinearGradientBrush>  
    </Button.Background>  
</Button>
```

Обратите внимание, что внутри дескрипторов `<Button>` и `</Button>` определена вложенная область по имени `<Button.Background>`, а в ней — специальный элемент `<LinearGradientBrush>`. (Пока не беспокойтесь о коде кисти; вы освоите графику WPF в главе 26.)

Любое свойство может быть установлено с использованием синтаксиса “свойство-элемент”, который всегда сводится к следующему шаблону:

```
<ОпределяющийКласс>  
  <ОпределяющийКласс.СвойствоОпределяющегоКласса>  
    <!-- Значение для свойства определяющего класса -->  
  </ОпределяющийКласс.СвойствоОпределяющегоКласса>  
</ОпределяющийКласс>
```

Хотя любое свойство может быть установлено с применением такого синтаксиса, указание значения в виде простой строки, когда подобное возможно, будет экономить время ввода. Например, вот гораздо более многословный способ установки свойства `Width` элемента `Button`:

```
<Button Height="50" Content="OK!"  
FontSize="20" Foreground="Yellow">  
...  
<Button.Width>  
100  
</Button.Width>  
</Button>
```

Понятие присоединяемых свойств XAML

В дополнение к синтаксису “свойство-элемент” в XAML поддерживается специальный синтаксис, используемый для установки значения *присоединяемого свойства*. По существу присоединяемое свойство позволяет дочернему элементу устанавливать значение свойства, которое определено в родительском элементе. Общий шаблон, которому нужно следовать, выглядит так:

```
<РодительскийЭлемент>  
  <ДочернийЭлемент РодительскийЭлемент.СвойствоРодительскогоЭлемента  
= "Значение">  
  </РодительскийЭлемент>
```

Самое распространенное применение синтаксиса присоединяемых свойств связано с позиционированием элементов пользовательского интерфейса внутри одного из классов диспетчеров компоновки (Grid, DockPanel и т.д.). Диспетчеры компоновки более подробно рассматриваются в главе 25, а пока введите в редакторе Kaxaml следующую разметку:

```
<Page  
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
  <Canvas Height="200" Width="200" Background="LightBlue">  
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20"  
         Width="20" Fill="DarkBlue"/>  
  </Canvas>  
</Page>
```

Здесь определен диспетчер компоновки Canvas, который содержит элемент Ellipse. Обратите внимание, что с помощью синтаксиса присоединяемых свойств элемент Ellipse способен информировать свой родительский элемент (Canvas) о том, где располагать позицию его левого верхнего угла.

В отношении присоединяемых свойств следует иметь в виду несколько моментов. Прежде всего, это не универсальный синтаксис, который может применяться к любому свойству любого родительского элемента. Скажем, приведенная далее разметка XAML содержит ошибку:

```
<!-- Попытка установки свойства Background в Canvas  
      через присоединяемое свойство. Ошибка! -->  
<Canvas Height="200" Width="200">  
  <Ellipse Canvas.Background="LightBlue"  
         Canvas.Top="40" Canvas.Left="90"  
         Height="20" Width="20" Fill="DarkBlue"/>  
</Canvas>
```

Присоединяемые свойства являются специализированной формой специфичной для WPF концепции, которая называется *свойством зависимости*. Если только свойс-

тво не было реализовано в весьма специальной манере, то его значение не может быть установлено с использованием синтаксиса присоединяемых свойств. Свойства зависимости подробно исследуются в главе 25.

На заметку! В Visual Studio имеется средство IntelliSense, которое отображает допустимые присоединяемые свойства, доступные для установки заданным элементом.

Понятие расширений разметки XAML

Как уже объяснялось, значения свойств чаще всего представляются в виде простой строки или через синтаксис “свойство-элемент”. Однако существует еще один способ указать значение атрибута XAML — применение *расширений разметки*. Расширения разметки позволяют анализатору XAML получать значение для свойства из выделенного внешнего класса. Это может обеспечить большие преимущества, поскольку для получения значений некоторых свойств требуется выполнение множества операторов кода.

Расширения разметки предлагают способ аккуратного расширения грамматики XAML новой функциональностью. Расширение разметки внутренне представлено как класс, производный от `MarkupExtension`. Следует отметить, что необходимость в построении специального расширения разметки возникает крайне редко. Тем не менее, некоторые ключевые слова XAML (вроде `x:Array`, `x:Null`, `x:Static` и `x:Type`) являются замаскированными расширениями разметки!

Расширение разметки помещается между фигурными скобками:

```
<Элемент УстанавливаемоеСвойство = "{РасширениеРазметки}"/>
```

Чтобы увидеть расширение разметки в действии, введите в редакторе XAML следующий код:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:CorLib="clr-namespace:System;assembly=mscorlib">
    <StackPanel>
        <!-- Расширение разметки Static позволяет получать значение
            статического члена класса -->
        <Label Content="{x:Static CorLib:Environment.OSVersion}"/>
        <Label Content="{x:Static CorLib:Environment.ProcessorCount}"/>
        <!-- Расширение разметки Type - это версия XAML
            операции typeof языка C# -->
        <Label Content="{x:Type Button}" />
        <Label Content="{x:Type CorLib:Boolean}" />
        <!-- Наполнение элемента ListBox массивом строк -->
        <ListBox Width="200" Height="50">
            <ListBox.ItemsSource>
                <x:Array Type="CorLib:String">
                    <CorLib:String>Sun Kil Moon</CorLib:String>
                    <CorLib:String>Red House Painters</CorLib:String>
                    <CorLib:String>Besnard Lakes</CorLib:String>
                </x:Array>
            </ListBox.ItemsSource>
        </ListBox>
    </StackPanel>
</Page>
```

Прежде всего, обратите внимание, что определение `<Page>` содержит новое объявление пространства имен XML, которое позволяет получать доступ к пространству имен System сборки mscorlib.dll. После установления этого пространства имен XML первым делом с помощью расширения разметки `x:Static` извлекаются значения свойств OSVersion и ProcessorCount класса System.Environment.

Расширение разметки `x:Type` обеспечивает доступ к описанию метаданных указанного элемента. Здесь содержимому элементов Label просто присваиваются полностью заданные имена типов Button и System.Boolean из WPF.

Наиболее интересная часть показанной выше разметки связана с элементом `ListBox`. Его свойство `ItemsSource` устанавливается в массив строк, полностью объявленный в разметке. Взгляните, каким образом расширение разметки `x:Array` позволяет указывать набор подэлементов внутри своей области действия:

```
<x:Array Type="CorLib:String">
  <CorLib:String>Sun Kil Moon</CorLib:String>
  <Corlib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

На заметку! Предыдущий пример XAML служит только для иллюстрации расширения разметки в действии. Как будет показано в главе 25, существуют гораздо более простые способы наполнения элементов управления `ListBox`.

На рис. 24.3 представлена разметка этого элемента `<Page>` в редакторе Kaxaml.

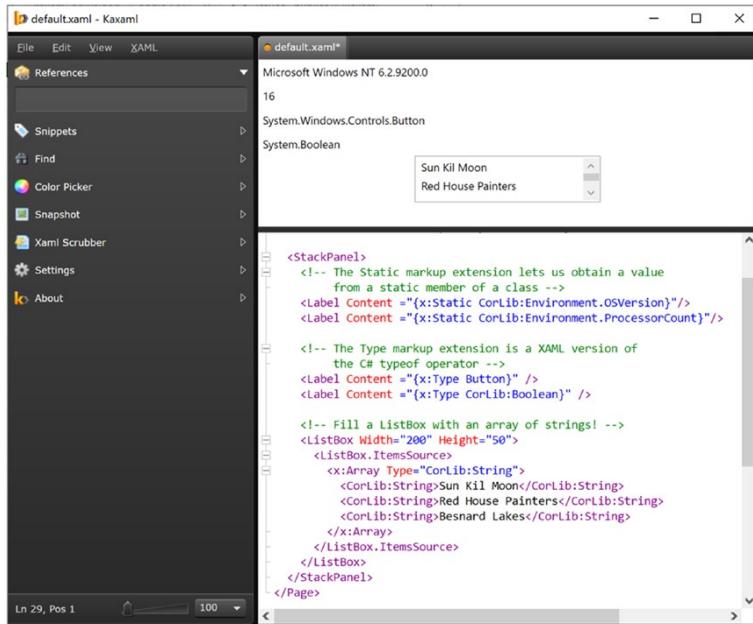


Рис. 24.3. Расширения разметки позволяют устанавливать значения через функциональность выделенного класса

Вы уже видели многочисленные примеры, которые демонстрировали основные аспекты синтаксиса XAML. Вы наверняка согласитесь, что XAML интересен своей возможностью описывать деревья объектов .NET в декларативной манере. Хотя это исключительно полезно при конфигурировании графических пользовательских интерфейсов, не забывайте о том, что с помощью XAML можно описывать любой тип из любой сборки при условии, что он является неабстрактным и содержит стандартный конструктор.

Построение приложений WPF с использованием Visual Studio

Давайте выясним, как Visual Studio может упростить создание приложений WPF. Хотя строить приложения WPF можно и с применением Visual Studio Code, в Visual Studio Code отсутствует поддержка соответствующих визуальных конструкторов. С другой стороны, благодаря развитой поддержке XAML среда Visual Studio обеспечивает более высокую продуктивность при создании приложений WPF.

На заметку! Далее будут представлены основные особенности применения Visual Studio для построения приложений WPF. В последующих главах при необходимости будут иллюстрироваться дополнительные аспекты этой IDE-среды.

Шаблоны проектов WPF

В диалоговом окне New Project (Новый проект) среды Visual Studio определен набор проектов приложений WPF, в том числе WPF App (Приложение WPF), WPF Custom Control Library (Библиотека специальных элементов управления WPF) и WPF User Control Library (Библиотека пользовательских элементов управления WPF). Создайте новый проект WPF App (.NET) по имени WpfTesterApp.

На заметку! При выборе шаблона проектов приложений WPF удостоверьтесь в том, что выбираете шаблон, который содержит в своем названии (.NET), но не (.NET Framework). Текущая версия .NET Core была переименована в просто .NET 5. Если вы выберете шаблон с (.NET Framework) в названии, то будете строить свое приложение, используя .NET Framework 4.x.

Кроме установки комплекта SDK в Microsoft .NET.Sdk вы получите начальные классы, производные от Window и Application, каждый из которых представлен с применением XAML и файла кода C#.

Панель инструментов и визуальный конструктор/редактор XAML

В Visual Studio имеется панель инструментов (открываемая через меню View (Вид)), которая содержит многочисленные элементы управления WPF. В верхней части панели расположены наиболее распространенные элементы управления, а в нижней части — все элементы управления (рис. 24.4).

С применением стандартной операции перетаскивания посредством мыши любой из элементов управления можно поместить на поверхность визуального конструктора элемента Window или перетащить его на область редактора разметки XAML внизу окна визуального конструктора.

После этого начальная разметка XAML сгенерируется автоматически. Давайте перетащим с помощью мыши элементы управления Button и Calendar на поверхность визуального конструктора. Обратите внимание на возможность изменения позиции и размера элементов управления (а также просмотрите результатирующую разметку XAML, генерируемую на основе изменений).

В дополнение к построению пользовательского интерфейса с использованием мыши и панели инструментов разметку можно также вводить вручную, применяя интегрированный редактор XAML. Как показано на рис. 24.5, вы получаете поддержку средства IntelliSense, которое помогает упростить написание разметки. Например, можете добавить свойство Background в открывающий элемент <Window>.

Посвятите некоторое время добавлению значений свойств напрямую в редакторе XAML. Обязательно освойте данный аспект визуального конструктора WPF.

Установка свойств с использованием окна Properties

После помещения нескольких элементов управления на поверхность визуального конструктора (или определения их в редакторе вручную) можно открыть окно Properties (Свойства) для установки значений свойств выделенного элемента управления, а также для создания связанных с ним обработчиков событий.

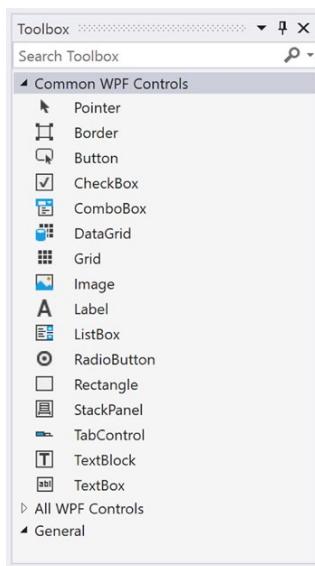


Рис. 24.4. Панель инструментов содержит элементы управления WPF, которые могут быть помещены на поверхность визуального конструктора

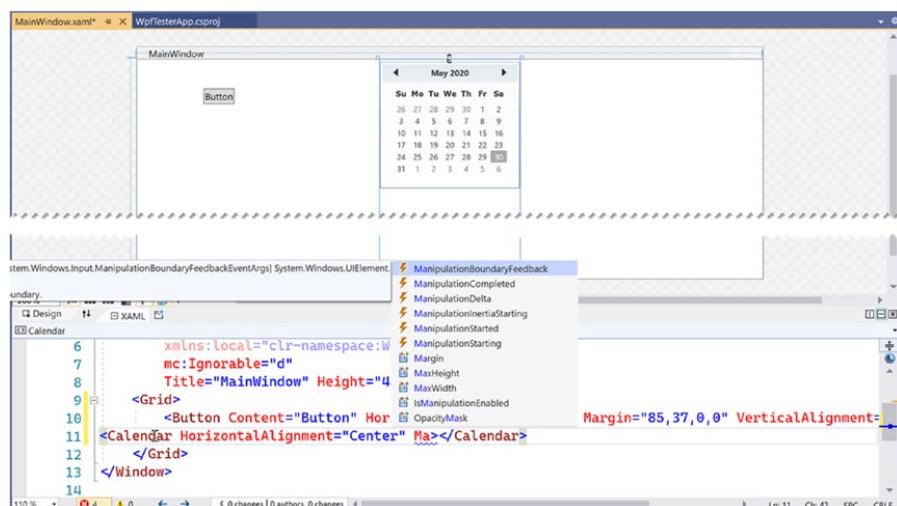


Рис. 24.5. Визуальный конструктор элемента Window

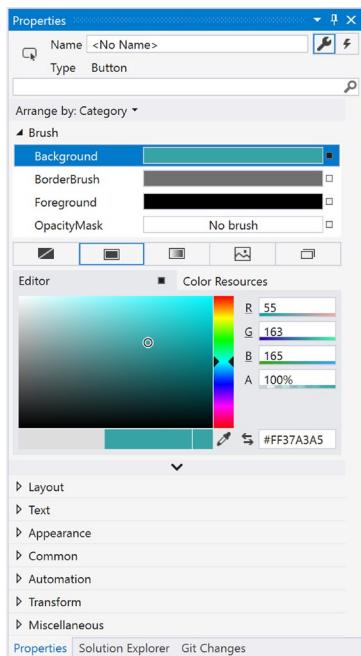


Рис. 24.6. Окно Properties может применяться для конфигурирования пользовательского интерфейса элемента управления WPF

В качестве простого примера выберите в визуальном конструкторе ранее добавленный элемент управления Button. С применением окна Properties измените цвет в свойстве Background элемента Button, используя встроенный редактор кистей (рис. 24.6); редактор кистей будет более подробно рассматриваться в главе 26 во время исследования графики WPF.

На заметку! В верхней части окна Properties имеется текстовая область, предназначенная для поиска. Чтобы быстро найти свойство, которое требуется установить, понадобится ввести его имя.

После завершения работы с редактором кистей имеет смысл взглянуть на генерированную разметку, которая может выглядеть так:

```
<Button x:Name="button" Content="Button"
        HorizontalAlignment="Left"
        Margin="10,10,0,0"
        VerticalAlignment="Top" Width="75">
    <Button.Background>
        <LinearGradientBrush EndPoint="0.5,1"
            StartPoint="0.5,0">
            <GradientStop Color="Black" Offset="0"/>
            <GradientStop Color="#FFE90E0E"
                Offset="1"/>
            <GradientStop Color="#FF1F4CE3"/>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

Обработка событий с использованием окна Properties

Для организации обработки событий, связанных с определенным элементом управления, также можно применять окно Properties, но на этот раз понадобится щелкнуть на кнопке Events (События), расположенной справа вверху окна (кнопка с изображением молнии). На поверхности визуального конструктора выберите элемент Button, если он еще не выбран, щелкните на кнопке Events в окне Properties и дважды щелкните на поле для события Click. Среда Visual Studio автоматически построит обработчик событий, имя которого имеет следующую общую форму:

ИмяЭлементаУправления_ИмяСобытия

Так как кнопка не была переименована, в окне Properties отображается генерированный обработчик событий по имени button_Click (рис. 24.7).

Кроме того, Visual Studio генерирует соответствующий обработчик события C# в файле кода для окна. В него можно поместить любой код, который должен выполняться, когда на кнопке произведен щелчок.

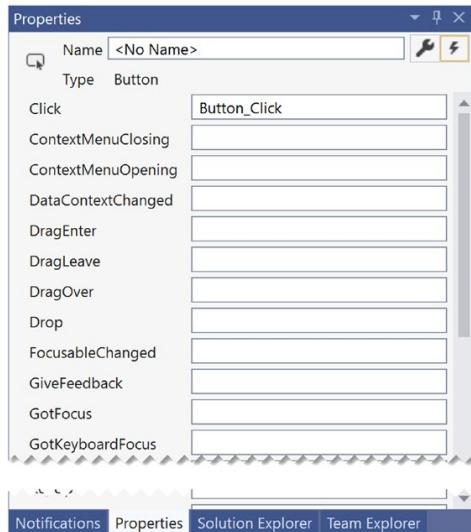


Рис. 24.7. Обработка событий с использованием окна Properties

В качестве простого примера добавьте следующий оператор кода:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked the button!");
    // Вы щелкнули на кнопке!
}
```

Обработка событий в редакторе XAML

Обрабатывать события можно и непосредственно в редакторе XAML. Например, поместите курсор мыши внутрь элемента `<Window>` и введите имя события `MouseMove`, а за ним знак равенства. Среда Visual Studio отобразит все совместимые обработчики из файла кода (если они существуют), а также пункт для создания нового обработчика событий (рис. 24.8).

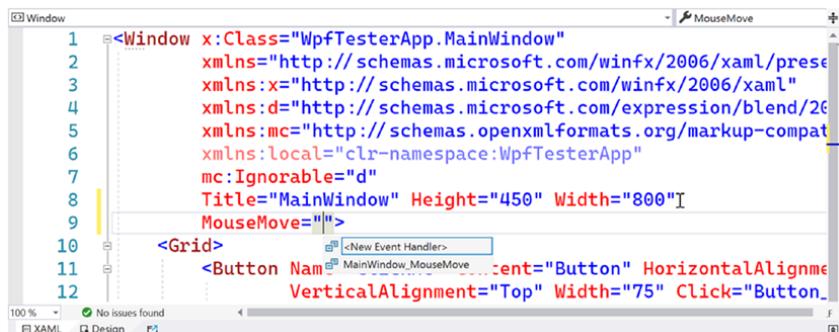


Рис. 24.8. Обработка событий с применением редактора XAML

Позвольте IDE-среде создать обработчик события MouseMove, введите следующий код и запустите приложение, чтобы увидеть результат:

```
private void MainWindow_MouseMove (object sender, MouseEventArgs e)
{
    this.Title = e.GetPosition(this).ToString();
}
```

На заметку! В главе 28 описаны паттерны MVVM и "Команда" (Command), которые являются гораздо лучшими способами обработки событий щелчков в корпоративных приложениях. Но если вас интересует только простое приложение, тогда обработка событий щелчков с помощью прямолинейного обработчика будет вполне приемлемой.

Окно Document Outline

Во время работы с любым основанным на XAML проектом вы определенно будете использовать значительный объем разметки для представления пользовательского интерфейса. Когда вы начнете сталкиваться с более сложной разметкой XAML, может оказаться удобной визуализация разметки для быстрого выбора элементов с целью редактирования в визуальном конструкторе Visual Studio.

В настоящее время ваша разметка довольно проста, т.к. было определено лишь несколько элементов управления внутри начального элемента `<Grid>`. Тем не менее, необходимо найти окно Document Outline (Схема документа), которое по умолчанию располагается в левой части окна IDE-среды (если обнаружить его не удается, то данное окно можно открыть через пункт меню View⇒Other Windows (Вид⇒Другие окна)). При

активном окне визуального конструктора XAML (не окне с файлом кода C#) в IDE-среде можно заметить, что в окне Document Outline отображаются вложенные элементы (рис. 24.9).

Этот инструмент также предоставляет способ временного скрытия заданного элемента (или набора элементов) на поверхности визуального конструктора, а также блокировки элементов с целью предотвращения их дальнейшего редактирования. В главе 25 вы увидите, что окно Document Outline предлагает много других возможностей для группирования выбранных элементов внутри новых диспетчеров компоновки (помимо прочих средств).

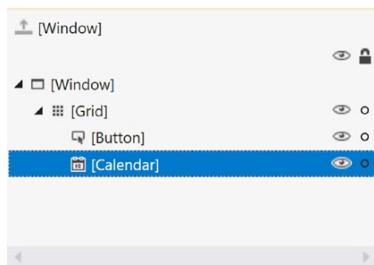


Рис. 24.9. Визуализация разметки XAML в окне Document Outline

Включение и отключение отладчика XAML

После запуска приложения на экране появляется окно MainWindow. Кроме того, можно также видеть интерактивный отладчик (рис. 24.10).

При желании отключить его понадобится найти настройки, касающиеся отладки XAML, на вкладке Tools⇒Options⇒Debugging⇒Hot Reload (Сервис⇒Параметры⇒Отладка⇒Горячая перезагрузка). Снятие отметки с верхнего флашка предотвращает перекрытие окон приложения окном отладчика (рис. 24.11).

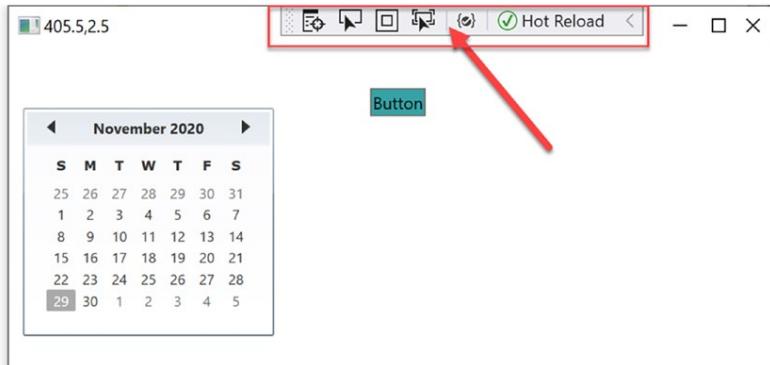


Рис. 24.10. Отладка пользовательского интерфейса XAML

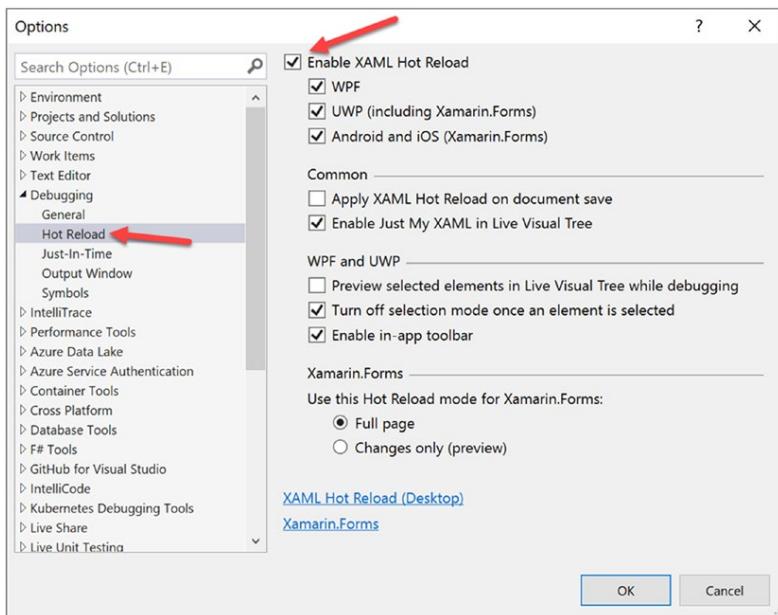


Рис. 24.11. Настройки, связанные с отладкой пользовательского интерфейса XAML

Исследование файла App.xaml

Как проект узнает, какое окно отображать? Еще большая интрига в том, что в результате исследования файлов кода, относящихся к приложению, метод Main() обнаружить не удастся. Вы уже знаете, что приложения обязаны иметь точку входа, так как же .NET Core становится известно, каким образом запускать приложение? К счастью, оба связующих элемента автоматически поддерживаются через шаблоны Visual Studio и инфраструктуру WPF.

Чтобы разгадать загадку, какое окно открывать, в файле App.xaml посредством разметки определен класс приложения. В дополнение к определениям пространств имен он определяет свойства приложения, такие как StartupUri, ресурсы уровня приложения (рассматриваемые в главе 27) и специфические обработчики для событий приложения вроде Startup и Exit. В StartupUri указано окно, подлежащее загрузке при запуске. Откройте файл App.xaml и проанализируйте разметку в нем:

```
<Application x:Class="WpfTesterApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfTesterApp"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

С применением визуального конструктора XAML и средства завершения кода Visual Studio добавьте обработчики для событий Startup и Exit. Обновленная разметка XAML должна выглядеть примерно так (изменение выделено полужирным):

```
<Application x:Class="WpfTesterApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfTesterApp"
    StartupUri="MainWindow.xaml"
    Startup="App_OnStartup" Exit="App_OnExit">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Содержимое файла App.xaml.cs должно быть похожим на приведенное ниже:

```
public partial class App : Application
{
    private void App_OnStartup(object sender, StartupEventArgs e)
    {
    }
    private void App_OnExit(object sender, ExitEventArgs e)
    {
    }
}
```

Обратите внимание, что класс помечен как частичный (`partial`). На самом деле все оконные классы в отдельном коде для файлов XAML помечаются как частичные. В этом-то и кроется решение вопроса, где находится метод `Main()`. Но сначала необходимо выяснить, что происходит при обработке файлов XAML утилитой msbuild.exe.

Отображение разметки XAML окна на код C#

Когда утилита msbuild.exe обрабатывает файл `*.csproj`, она создает для каждого файла XAML в проекте три файла: `*.g.cs` (где `g` означает `autogenerated` (автоматически генерированный)), `*.g.i.cs` (где `i` означает `IntelliSense`) и `*.baml` (для BAML (Binary Application Markup Language — двоичный язык разметки приложений)). Такие файлы сохраняются в каталоге `\obj\Debug` (и могут просматриваться в окне Solution Explorer за счет щелчка на кнопке `Show All Files` (Показать все файлы)).

Чтобы их увидеть, может потребоваться щелкнуть на кнопке Refresh (Обновить) в окне Solution Explorer, т.к. они не являются частью фактического проекта, а представляют собой артефакты построения.

Чтобы сделать процесс более осмысленным, элементам управления полезно назначить имена. Назначьте имена элементам управления Button и Calendar, как показано ниже:

```
<Button Name="ClickMe" Content="Button" HorizontalAlignment="Left"
    Margin="10,10,0,0"
    VerticalAlignment="Top" Width="75" Click="Button_Click">
    // Для краткости разметка не показана.
</Button>
<Calendar Name="MyCalendar" HorizontalAlignment="Left"
    Margin="10,41,0,0" VerticalAlignment="Top"/>
```

Теперь повторно скомпилируйте решение (или проект) и обновите файлы в окне Solution Explorer. Если открыть файл MainWindow.g.cs в текстовом редакторе, то внутри обнаружится класс по имени MainWindow, который расширяет базовый класс Window. Имя данного класса является прямым результатом действия атрибута x:Class в начальном дескрипторе <Window>.

В классе MainWindow определена закрытая переменная-член типа bool (с именем _contentLoaded), которая не была напрямую представлена в разметке XAML. Указанный член данных используется для того, чтобы определить (и гарантировать) присваивание содержимого окна только один раз. Класс также содержит переменную-член типа System.Windows.Controls.Button по имени ClickMe. Имя элемента управления основано на значении атрибута x:Name в открывающем объявлении <Button>. В классе не будет присутствовать переменная для элемента управления Calendar. Причина в том, что утилита msbuild.exe создает переменную для каждого именованного элемента управления в разметке XAML, который имеет связанный код в отдельном коде. Когда такого кода нет, потребность в переменной отпадает. Чтобы еще больше запутать ситуацию, если бы элементу управления Button не назначалось имя, то и для него не было бы предусмотрено переменной. Это часть магии WPF, которая связана с реализацией интерфейса IComponentConnector.

Сгенерированный компилятором класс также явно реализует интерфейс IComponentConnector из WPF, определенный в пространстве имен System.Windows.Markup. В интерфейсе IComponentConnector имеется единственный метод Connect(), который реализован для подготовки каждого элемента управления, определенного в разметке, и обеспечения логики событий, как указано в исходном файле MainWindow.xaml. Можно заметить обработчик, настроенный для события щелчка на кнопке ClickMe. Перед завершением метода переменная-член _contentLoaded устанавливается в true. Вот как выглядит данный метод:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
    object target)
{
    switch (connectionId)
    {
        case 1:
            this.ClickMe = ((System.Windows.Controls.Button)(target));
            #line 11 "..\..\MainWindow.xaml"
            this.ClickMe.Click +=
                new System.Windows.RoutedEventHandler(this.Button_Click);
```

```

        #line default
        #line hidden
        return;
    }
    this._contentLoaded = true;
}

```

Чтобы продемонстрировать влияние неименованных элементов управления на код, добавьте к календарю обработчик события SelectedDatesChanged. Перекомпилируйте приложение, обновите файлы и заново загрузите файл MainWindow.g.cs. Теперь в методе Connect() присутствует следующий блок кода:

```

#line 20 "...\\MainWindow.xaml"
this.MyCalendar.SelectedDatesChanged += new
    System.EventHandler<System.Windows.Controls.SelectionChangedEventArgs>(
        this.MyCalendar_OnSelectedDatesChanged);

```

Он сообщает инфраструктуре о том, что элементу управления в строке 20 файла XAML назначен обработчик события SelectedDatesChanged, как показано в предыдущем коде.

Наконец, класс MainWindow определяет и реализует метод по имени InitializeComponent Component(). Вы могли бы ожидать, что данный метод содержит код, который настраивает внешний вид и поведение каждого элемента управления, устанавливая его разнообразные свойства (Height, Width, Content и т.д.). Однако это совсем не так! Как тогда элементы управления получают корректный пользовательский интерфейс? Логика в методе InitializeComponent Component() выясняет местоположение встроенного в сборку ресурса, который именован идентично исходному файлу *.xaml:

```

public void InitializeComponent() {
    if (_contentLoaded) {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater =
        new System.Uri("/WpfTesterApp;component/mainwindow.xaml",
            System.UriKind.Relative);
    #line 1 "...\\MainWindow.xaml"
    System.Windows.Application.LoadComponent(this, resourceLocater);
    #line default
    #line hidden
}

```

Здесь возникает вопрос: что собой представляет этот встроенный ресурс?

Роль BAML

Как и можно было предположить, формат BAML является компактным двоичным представлением исходных данных XAML. Файл *.baml встраивается в виде ресурса (через генерированный файл *.g.resources) в скомпилированную сборку. Ресурс BAML содержит все данные, необходимые для настройки внешнего вида и поведения виджетов пользовательского интерфейса (т.е. свойств вроде Height и Width).

Здесь важно понимать, что приложение WPF содержит внутри себя двоичное представление (BAML) разметки. Во время выполнения ресурс BAML извлекается из контейнера ресурсов и применяется для настройки внешнего вида и поведения всех окон и элементов управления.

В добавок запомните, что имена таких двоичных ресурсов *идентичны* именам написанных автономных файлов *.xaml. Тем не менее, отсюда вовсе не следует необходимости распространения файлов *.xaml вместе со скомпилированной программой WPF. Если только не строится приложение WPF, которое должно динамически загружать и анализировать файлы *.xaml во время выполнения, то поставлять исходную разметку никогда не придется.

Разгадывание загадки Main()

Теперь, когда известно, как работает процесс msbuild.exe, откройте файл App.g.cs. В нем обнаружится автоматически сгенерированный метод Main(), который инициализирует и запускает ваш объект приложения:

```
public static void Main() {
    WpfTesterApp.App app = new WpfTesterApp.App();
    app.InitializeComponent();
    app.Run();
}
```

Метод InitializeComponent() конфигурирует свойства приложения, включая StartupUri и обработчики событий Startup и Exit:

```
public void InitializeComponent() {
    #line 5 "..\..\App.xaml"
    this.Startup +=
        new System.Windows.StartupEventHandler(this.App_OnStartup);
    #line default
    #line hidden
    #line 5 "..\..\App.xaml"
    this.Exit += new System.Windows.ExitEventHandler(this.App_OnExit);
    #line default
    #line hidden
    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml",
        System.UriKind.Relative);
    #line default
    #line hidden
}
```

Взаимодействие с данными уровня приложения

Вспомните, что в классе Application имеется свойство по имени Properties, которое позволяет определить коллекцию пар “имя/значение” посредством индексатора типа. Поскольку этот индексатор предназначен для оперирования на типе System.Object, в коллекцию можно сохранять элементы любого вида (в том числе экземпляры специальных классов) с целью последующего извлечения по дружественному имени. С использованием такого подхода легко разделять данные между всеми окнами в приложении WPF.

В целях иллюстрации вы обновите текущий обработчик события Startup, чтобы он проверял входящие аргументы командной строки на присутствие значения /GODMODE (распространенный мошеннический код во многих играх). Если оно найдено, тогда значение bool по имени GodMode внутри коллекции свойств устанавливается в true (в противном случае оно устанавливается в false).

Звучит достаточно просто, но как передать обработчику события `Startup` входные аргументы командной строки (обычно получаемые методом `Main()`)? Один из подходов предусматривает вызов статического метода `Environment.GetCommandLineArgs()`. Однако те же самые аргументы автоматически добавляются во входной параметр `StartupEventArgs` и доступны через свойство `Args`. Ниже приведена первая модификация текущей кодовой базы:

```
private void App_OnStartup(object sender, StartupEventArgs e)
{
    Application.Current.Properties["GodMode"] = false;
    // Проверить входные аргументы командной строки
    // на предмет наличия флага /GODMODE.
    foreach (string arg in e.Args)
    {
        if (arg.Equals("/godmode", StringComparison.OrdinalIgnoreCase))
        {
            Application.Current.Properties["GodMode"] = true;
            break;
        }
    }
}
```

Данные уровня приложения доступны из любого места внутри приложения WPF. Для обращения к ним потребуется лишь получить точку доступа к глобальному объекту приложения (через `Application.Current`) и просмотреть коллекцию. Например, обработчик события `Click` для кнопки можно было бы изменить следующим образом:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Указал ли пользователь /godmode?
    if ((bool)Application.Current.Properties["GodMode"])
    {
        MessageBox.Show("Cheater!");           // Мошенник!
    }
}
```

Если теперь ввести аргумент командной строки `/godmode` на вкладке `Debug` (Отладка) в окне свойств проекта и запустить программу, то отобразится окно сообщения и программа завершится. Можно также запустить программу из командной строки с помощью показанной ниже команды (предварительно открыв окно командной строки и перейдя в каталог `bin/debug`):

```
WpfAppAllCode.exe /godmode
```

Отобразится окно сообщения и программа завершится.

На заметку! Вспомните, что аргументы командной строки можно указывать внутри Visual Studio. Нужно просто дважды щелкнуть на значке `Properties` (Свойства) в окне `Solution Explorer`, в открывшемся диалоговом окне перейти на вкладку `Debug` (Отладка) и ввести `/godmode` в поле `Command line arguments` (Аргументы командной строки).

Обработка закрытия объекта `Window`

Конечные пользователи могут завершать работу окна с помощью многочисленных встроенных приемов уровня системы (например, щелкнув на кнопке закрытия

Х внутри рамки окна) или вызывав метод `Close()` в ответ на некоторое действие с интерактивным элементом (скажем, выбор пункта меню `File⇒Exit` (Файл⇒Выход)). Инфраструктура WPF предлагает два события, которые можно перехватывать для выяснения, действительно ли пользователь намерен закрыть окно и удалить его из памяти. Первое такое событие — `Closing`, которое работает в сочетании с делегатом `CancelEventHandler`.

Указанный делегат ожидает целевые методы, принимающие тип `System.ComponentModel.CancelEventArgs` во втором параметре. Класс `CancelEventArgs` предоставляет свойство `Cancel`, установка которого в `true` предотвращает фактическое закрытие окна (что удобно, когда пользователю должен быть задан вопрос о том, на самом ли деле он желает закрыть окно или сначала нужно сохранить результаты проделанной работы). Если пользователь действительно хочет закрыть окно, тогда свойство `CancelEventArgs.Cancel` можно установить в `false` (стандартное значение). В итоге генерируется событие `Closed` (которое работает с делегатом `System.EventHandler`), представляющее собой точку, где окно полностью и безвозвратно готово к закрытию.

Модифицируйте класс `MainWindow` для обработки упомянутых двух событий, добавив в текущий код конструктора такие операторы:

```
public MainWindow()
{
    InitializeComponent();
    this.Closed+=MainWindow_Closed;
    this.Closing += MainWindow_Closing;
}
```

Теперь реализуйте соответствующие обработчики событий:

```
private void MainWindow_Closing(object sender,
                               System.ComponentModel.CancelEventArgs e)
{
    // Выяснить, на самом ли деле пользователь хочет закрыть окно.
    string msg = "Do you want to close without saving?";
    MessageBoxResult result = MessageBox.Show(msg,
        "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
    if (result == MessageBoxResult.No)
    {
        // Если пользователь не желает закрывать окно, тогда отменить закрытие
        e.Cancel = true;
    }
}
private void MainWindow_Closed(object sender, EventArgs e)
{
    MessageBox.Show("See ya!");
}
```

Запустите программу и попробуйте закрыть окно, щелкнув либо на значке X в правом верхнем углу окна, либо на кнопке. Должно появиться диалоговое окно с запросом подтверждения. Щелчок на кнопке Yes (Да) приведет к отображению окна с прощальным сообщением, а щелчок на кнопке No (Нет) оставит окно в памяти.

Перехват событий мыши

Инфраструктура WPF предоставляет несколько событий, которые можно перехватывать, чтобы взаимодействовать с мышью. В частности, базовый класс `UIElement` определяет такие связанные с мышью события, как `MouseMove`, `MouseUp`, `MouseDown`, `MouseEnter`, `MouseLeave` и т.д.

В качестве примера обработайте событие `MouseMove`. Это событие работает в сочетании с делегатом `System.Windows.Input.MouseEventHandler`, который ожидает, что его целевой метод будет принимать во втором параметре объект типа `System.Windows.Input.MouseEventArgs`. С применением класса `MouseEventArgs` можно извлекать позицию (x, y) курсора мыши и другие важные детали. Взгляните на следующее неполное определение:

```
public class MouseEventArgs : InputEventArgs
{
    ...
    public Point GetPosition(IIInputElement relativeTo);
    public MouseButtonState LeftButton { get; }
    public MouseButtonState MiddleButton { get; }
    public MouseDevice MouseDevice { get; }
    public MouseButtonState RightButton { get; }
    public StylusDevice StylusDevice { get; }
    public MouseButtonState XButton1 { get; }
    public MouseButtonState XButton2 { get; }
}
```

На заметку! Свойства `XButton1` и `XButton2` позволяют взаимодействовать с "расширенными кнопками мыши" (вроде кнопок "вперед" и "назад", которые имеются в некоторых устройствах). Они часто используются для взаимодействия с хронологией навигации браузера, чтобы перемещаться между посещенными страницами.

Метод `GetPosition()` позволяет получать значение (x, y) относительно какого-то элемента пользовательского интерфейса в окне. Если интересует позиция относительно активного окна, то нужно просто передать `this`. Обеспечьте обработку события `MouseMove` в конструкторе класса `MainWindow`:

```
public MainWindow(string windowTitle, int height, int width)
{
    ...
    this.MouseEventHandler += MainWindow_MouseMove;
}
```

Ниже приведен обработчик события `MouseMove`, который отобразит местоположение курсора мыши в области заголовка окна (обратите внимание, что возвращенный тип `Point` транслируется в строковое значение посредством вызова `ToString()`):

```
private void MainWindow_MouseMove(object sender,
    System.Windows.Input.MouseEventArgs e)
{
    // Отобразить в заголовке окна текущую позицию (x, y) курсора мыши.
    this.Title = e.GetPosition(this).ToString();
}
```

Перехват событий клавиатуры

Обработка клавиатурного ввода для окна, на котором находится фокус, тоже очень проста. В классе `UIElement` определено несколько событий, которые можно перехватывать для отслеживания нажатий клавиш клавиатуры на активном элементе (например, `KeyUp` и `KeyDown`). События `KeyUp` и `KeyDown` работают с делегатом

`System.Windows.Input.KeyEventArgs`, который ожидает во втором параметре тип `KeyEventArgs`, определяющий набор важных открытых свойств:

```
public class KeyEventArgs : KeyboardEventArgs
{
    ...
    public bool IsDown { get; }
    public bool IsRepeat { get; }
    public bool IsToggled { get; }
    public bool IsUp { get; }
    public Key Key { get; }
    public KeyStates KeyStates { get; }
    public Key SystemKey { get; }
}
```

Чтобы проиллюстрировать организацию обработки события `KeyDown` в конструкторе `MainWindow` (как делалось для предыдущих событий), можно реализовать обработчик события, который изменяет содержимое кнопки на информацию о текущей нажатой клавише:

```
private void MainWindowOn_KeyDown(object sender,
                                 System.Windows.Input.KeyEventArgs e)
{
    // Отобразить на кнопке информацию о нажатой клавише.
    ClickMe.Content = e.Key.ToString();
}
```

К настоящему моменту WPF может показаться всего лишь очередной инфраструктурой для построения графических пользовательских интерфейсов, которая предлагает (в большей или меньшей степени) те же самые службы, что и Windows Forms, MFC или VB6. Если бы это было именно так, тогда возникает вопрос о смысле наличия еще одного инструментального набора, ориентированного на создание пользовательских интерфейсов. Чтобы реально оценить уникальность WPF, потребуется освоить основанную на XML грамматику — XAML.

Резюме

Инфраструктура Windows Presentation Foundation (WPF) представляет собой набор инструментов для построения пользовательских интерфейсов, появившийся в версии .NET 3.0. Основная цель WPF заключается в интеграции и унификации множества ранее разрозненных настольных технологий (двумерная и трехмерная графика, разработка окон и элементов управления и т.п.) в единую программную модель. Помимо этого в приложениях WPF обычно применяется язык XAML, который позволяет определять внешний вид и поведение элементов WPF через разметку.

Вспомните, что язык XAML позволяет описывать деревья объектов .NET с использованием декларативного синтаксиса. Во время исследования XAML в данной главе вы узнали о нескольких новых фрагментах синтаксиса, включая синтаксис “свойство-элемент” и присоединяемые свойства, а также о роли преобразователей типов и расширений разметки XAML.

Разметка XAML является ключевым аспектом любого приложения WPF производственного уровня. В финальном примере главы было построено приложение WPF, которое продемонстрировало многие концепции, обсужденные в главе. В последующих главах эти и многие другие концепции будут рассматриваться более подробно.

ГЛАВА 25

Элементы управления, компоновки, события и привязка данных в WPF

В главе 24 была представлена основа программной модели WPF, включая классы `Window` и `Application`, грамматику XAML и использование файлов кода. Кроме того, в ней было дано введение в процесс построения приложений WPF с применением визуальных конструкторов IDE-среды Visual Studio. В настоящей главе вы углубитесь в конструирование более сложных графических пользовательских интерфейсов с использованием нескольких новых элементов управления и диспетчеров компоновки, а также по ходу дела выясните дополнительные возможности визуального конструктора WPF в Visual Studio.

Здесь будут рассматриваться некоторые важные темы, связанные с элементами управления WPF, такие как программная модель привязки данных и применение команд управления. Вы узнаете, как работать с интерфейсами Ink API и Documents API, которые позволяют получать ввод от пера (или мыши) и создавать форматированные документы с использованием протокола XML Paper Specification.

Обзор основных элементов управления WPF

Если вы не являетесь новичком в области построения графических пользовательских интерфейсов, то общее назначение большинства элементов управления WPF не должно вызывать много вопросов. Независимо от того, какой набор инструментов для создания графических пользовательских интерфейсов вы применяли в прошлом (например, VB6, MFC, Java AWT/Swing, Windows Forms, GTK+/GTK# и т.п.), основные элементы управления WPF, перечисленные в табл. 25.1, вероятно покажутся знакомыми.

На заметку! Целью настоящей главы не является рассмотрение абсолютно всех членов каждого элемента управления WPF. Взамен вы получаете обзор разнообразных элементов управления с упором на лежащую в основе программную модель и ключевые службы, общие для большинства элементов управления WPF.

Таблица 25.1. Основные элементы управления WPF

Категория элементов управления WPF	Примеры членов	Описание
Основные элементы управления для пользовательского ввода	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	Инфраструктура WPF предлагает полное семейство элементов управления, которые можно задействовать при построении пользовательских интерфейсов
Боковые элементы окон и элементов управления	Menu,ToolBar, StatusBar, ToolTip, ProgressBar	Эти элементы пользовательского интерфейса служат для декорирования рамки объекта Window компонентами для ввода (наподобие Мени) и элементами информирования пользователя (скажем, StatusBar и ToolTip)
Элементы управления мультимедиа	Image, MediaElement, SoundPlayerAction	Эти элементы управления предоставляют поддержку воспроизведения аудио-/видеороликов и вывода изображений
Элементы управления компоновкой	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	Инфраструктура WPF предлагает множество элементов управления, которые позволяют группировать и организовывать другие элементы в целях управления компоновкой

Элементы управления для работы с Ink API

В дополнение к общепринятым элементам управления WPF, упомянутым в табл. 25.1, инфраструктура WPF определяет элементы управления для работы с интерфейсом Ink API. Данный аспект разработки WPF полезен при построении приложений для Tablet PC, т.к. он позволяет захватывать ввод от пера. Тем не менее, это вовсе не означает, что стандартное настольное приложение не может задействовать Ink API, поскольку те же самые элементы управления могут работать с вводом от мыши.

Пространство имен System.Windows.Ink из сборки PresentationCore.dll содержит разнообразные поддерживающие типы Ink API (например, Stroke и StrokeCollection). Однако большинство элементов управления Ink API (вроде InkCanvas и InkPresenter) упакованы вместе с общими элементами управления WPF в пространстве имен System.Windows.Controls внутри сборки PresentationFramework.dll. Мы будем работать с интерфейсом Ink API позже в главе.

Элементы управления для работы с документами WPF

В добавок инфраструктура WPF предоставляет элементы управления для расширенной обработки документов, позволяя строить приложения, которые включают функциональность в стиле Adobe PDF. С применением типов из пространства имен `System.Windows.Documents` (также находящегося в сборке `PresentationFramework.dll`) можно создавать готовые к печати документы, которые поддерживают масштабирование, поиск, пользовательские аннотации ("клейкие" заметки) и другие развитые средства работы с текстом.

Тем не менее, "за кулисами" элементы управления документов не используют API-интерфейсы Adobe PDF, а взамен работают с API-интерфейсом XML Paper Specification (XPS). Конечные пользователи никакой разницы не заметят, потому что документы PDF и XPS имеют практически идентичный вид и поведение. В действительности доступно множество бесплатных утилит, которые позволяют выполнять преобразования между указанными двумя файловыми форматами на лету. Из-за ограничений по объему такие элементы управления в текущем издании не рассматриваются.

Общие диалоговые окна WPF

Инфраструктура WPF также предлагает несколько общих диалоговых окон, таких как `OpenFileDialog` и `SaveFileDialog`, которые определены в пространстве имен `Microsoft.Win32` внутри сборки `PresentationFramework.dll`. Работа с любым из указанных диалоговых окон сводится к созданию объекта и вызову метода `ShowDialog()`:

```
using Microsoft.Win32;
// Для краткости код не показан.
private void btnShowDlg_Click(object sender, RoutedEventArgs e)
{
    // Отобразить диалоговое окно сохранения файла.
    SaveFileDialog saveDlg = new SaveFileDialog();
    saveDlg.ShowDialog();
}
```

Как и можно было ожидать, в этих классах поддерживаются разнообразные члены, которые позволяют устанавливать фильтры файлов и пути к каталогам, а также получать доступ к выбранным пользователем файлам. Некоторые диалоговые окна применяются в последующих примерах; кроме того, будет показано, как строить специальные диалоговые окна для получения пользовательского ввода.

Краткий обзор визуального конструктора WPF в Visual Studio

Большинство стандартных элементов управления WPF упаковано в пространство имен `System.Windows.Controls` внутри сборки `PresentationFramework.dll`. При построении приложения WPF в Visual Studio множество общих элементов управления находится в панели инструментов при условии, что активным окном является визуальный конструктор WPF.

Подобно другим инфраструктурам для построения пользовательских интерфейсов в Visual Studio такие элементы управления можно перетаскивать на поверхность визуального конструктора WPF и конфигурировать их в окне `Properties` (Свойства), как

было показано в главе 24. Хотя Visual Studio генерирует приличный объем разметки XAML автоматически, нет ничего необычного в том, чтобы затем редактировать разметку вручную. Давайте рассмотрим основы.

Работа с элементами управления WPF в Visual Studio

Вы можете вспомнить из главы 24, что после помещения элемента управления WPF на поверхность визуального конструктора Visual Studio в окне Properties (или прямо в разметке XAML) необходимо установить свойство `x:Name`, т.к. это позволяет обращаться к объекту в связанном файле кода C#. Кроме того, на вкладке Events (События) окна Properties можно генерировать обработчики событий для выбранного элемента управления. Таким образом, с помощью Visual Studio можно было бы генерировать следующую разметку для простого элемента управления Button:

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140"
        Click="btnMyButton_Click" />
```

Здесь свойство `Content` элемента Button устанавливается в простую строку "Click Me!". Однако благодаря модели содержимого элементов управления WPF можно создать элемент Button со следующим сложным содержимым:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
        Click="btnMyButton_Click">
    <Button.Content>
        <StackPanel Height="95" Width="128" Orientation="Vertical">
            <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
            <Label Width="59" FontSize="20" Content="Click!" Height="36" />
        </StackPanel>
    </Button.Content>
</Button>
```

Вы можете также вспомнить, что непосредственным дочерним элементом производного от `ContentControl` класса является предполагаемое содержимое, а потому при указании сложного содержимого определять область `Button.Content` явно не требуется. Можно было бы написать такую разметку:

```
<Button x:Name="btnMyButton" Height="121" Width="156"
        Click="btnMyButton_Click">
    <StackPanel Height="95" Width="128" Orientation="Vertical">
        <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
        <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
</Button>
```

В любом случае свойство `Content` кнопки устанавливается в элемент `StackPanel` со связанными элементами. Создавать сложное содержимое подобного рода можно также с применением визуального конструктора Visual Studio. После определения диспетчера компоновки для элемента управления содержимым его можно выбирать в визуальном конструкторе в качестве целевого компонента, на который будут перетаскиваться внутренние элементы управления. Каждый из них можно редактировать в окне Properties. Если окно Properties использовалось для обработки события `Click` элемента управления Button (как было показано в предшествующих объявлениях XAML), то IDE-среда генерирует пустой обработчик события, куда можно будет добавить специальный код, например:

```

private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked the button!");
    // Вы щелкнули на кнопке!
}

```

Работа с окном Document Outline

В главе 24 вы узнали, что при проектировании элемента управления WPF со сложным содержимым удобно пользоваться окном Document Outline (Схема документа) в Visual Studio (открываемое через меню View⇒Other Windows (Вид⇒Другие окна)). Для создаваемого элемента Window отображается логическое дерево XAML, а щелчок на любом узле в дереве приводит к его автоматическому выбору визуальном конструкторе и в редакторе XAML для редактирования.

В текущей версии Visual Studio окно Document Outline имеет несколько дополнительных средств, которые вы можете счесть полезными. Справа от любого узла находится значок, напоминающий глазное яблоко. Щелчок на нем позволяет скрывать или отображать элемент в визуальном конструкторе, что оказывается удобным, когда необходимо сосредоточить внимание на отдельном сегменте, подлежащем редактированию (следует отметить, что элемент будет скрыт только на поверхности визуального конструктора, но *не во время выполнения*).

Справа от значка с глазным яблоком есть еще один значок, который позволяет блокировать элемент в визуальном конструкторе. Как и можно было догадаться, это удобно, когда нужно воспрепятствовать случайному изменению разметки XAML для заданного элемента вами или коллегами по разработке. На самом деле блокировка элемента делает его допускающим только чтение на этапе проектирования (но вы можете изменять состояние объекта во время выполнения).

Управление компоновкой содержимого с использованием панелей

Приложение WPF неизменно содержит определенное количество элементов пользовательского интерфейса (например, элементов ввода, графического содержимого, систем меню и строк состояния), которые должны быть хорошо организованы внутри разнообразных окон. После размещения элементов пользовательского интерфейса необходимо гарантировать их запланированное поведение, когда конечный пользователь изменяет размер окна или его части (как в случае окна с разделителем). Чтобы обеспечить сохранение элементами управления WPF своих позиций внутри окна, в котором они находятся, можно использовать множество типов панелей (также известных как диспетчеры компоновки).

По умолчанию новый WPF-элемент Window, созданный с помощью Visual Studio, будет применять диспетчер компоновки типа Grid (вскоре мы опишем его более подробно). Тем не менее, пока что рассмотрим элемент Window без каких-либо объявленных диспетчеров компоновки:

```

<Window x:Class="MyWPFApp.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Fun with Panels!" Height="285" Width="325">
</Window>

```

Когда элемент управления объявляется прямо внутри окна, в котором не используются панели, он позиционируется по центру контейнера. Рассмотрим показанное далее простое объявление окна, содержащего единственный элемент управления Button. Независимо от того, как изменяются размеры окна, этот виджет пользовательского интерфейса всегда будет находиться на равном удалении от всех четырех границ клиентской области. Размер элемента Button определяется установленными значениями его свойств Height и Width.

```
<!-- Эта кнопка всегда находится в центре окна -->
<Window x:Class="MyWPFApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="285" Width="325">
    <Button x:Name="btnOK" Height = "100"
        Width="80" Content="OK"/>
</Window>
```

Также вспомните, что попытка помещения внутрь области Window сразу нескольких элементов вызовет ошибки разметки и компиляции. Причина в том, что свойству Content окна (или по существу любого потомка ContentControl) может быть присвоен только один объект. Следовательно, приведенная далее разметка XAML приведет к ошибкам разметки и компиляции:

```
<!-- Ошибка! Свойство Content неявно устанавливается более одного раза! -->
<Window x:Class="MyWPFApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="285" Width="325">
    <!-- Ошибка! Два непосредственных дочерних элемента в <Window>! -->
    <Label x:Name="lblInstructions" Width="328" Height="25"
        FontSize="15" Content="Enter Information"/>
    <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

Понятно, что от окна, допускающего наличие только одного элемента управления, мало толку. Когда окно должно содержать несколько элементов, их потребуется расположить внутри любого числа панелей. В панель будут помещены все элементы пользовательского интерфейса, которые представляют окно, после чего сама панель выступает в качестве единственного объекта, присваиваемого свойству Content окна.

Пространство имен System.Windows.Controls предлагает многочисленные панели, каждая из которых по-своему обслуживает внутренние элементы. С помощью панелей можно устанавливать поведение элементов управления при изменении размеров окна пользователем — будут они оставаться в тех же местах, где были размещены на этапе проектирования, располагаться свободным потоком слева направо или сверху вниз и т.д.

Элементы управления типа панелей также разрешено помещать внутрь других панелей (например, элемент управления DockPanel может содержать StackPanel со своими элементами), чтобы обеспечить высокую гибкость и степень управления. В табл. 25.2 кратко описаны некоторые распространенные элементы управления типа панелей WPF.

Таблица 25.2. Основные элементы управления типа панелей WPF

Элемент управления типа панели	Описание
Canvas	Предоставляет классический режим размещения содержимого. Элементы остаются в точности там, куда были помещены на этапе проектирования
DockPanel	Привязывает содержимое к указанной стороне панели (Top (верхняя), Bottom (нижняя), Left (левая) или Right (правая))
Grid	Располагает содержимое внутри серии ячеек, поддерживаемых внутри табличной сетки
StackPanel	Укладывает содержимое вертикально или горизонтально, как регламентируется свойством Orientation
WrapPanel	Позиционирует содержимое слева направо, перенося его на следующую строку по достижении границы панели. Дальнейшее упорядочение происходит последовательно сверху вниз или слева направо в зависимости от значения свойства Orientation

В последующих нескольких разделах вы узнаете, как применять распространенные типы панелей, копируя заранее определенную разметку XAML в редактор Xaml, который был установлен в главе 24. Все необходимые файлы XAML находятся в подкаталоге PanelMarkup внутри Chapter_25. Во время работы с Xaml для эмуляции изменения размеров окна нужно изменить высоту или ширину элемента Page в разметке.

Позиционирование содержимого внутри панелей Canvas

При наличии опыта работы с Windows Forms панель Canvas вероятно покажется наиболее привычной, т.к. она делает возможным абсолютное позиционирование содержимого пользовательского интерфейса. Если конечный пользователь изменяет размер окна, делая его меньше, чем размер компоновки, обслуживаемой панелью Canvas, то внутреннее содержимое будет невидимым до тех пор, пока контейнер не увеличится до размера, равного или превышающего размер области Canvas.

Чтобы добавить содержимое к Canvas, сначала понадобится определить требуемые элементы управления внутри области между открывающим и закрывающим дескрипторами Canvas. Затем для каждого элемента управления необходимо указать левый верхний угол с использованием свойств Canvas.Top и Canvas.Left; именно здесь должна начинаться визуализация. Правый нижний угол каждого элемента управления можно задать неявно, устанавливая свойства Canvas.Height и Canvas.Width, либо явно с применением свойств Canvas.Right и Canvas.Bottom.

Для демонстрации Canvas в действии откройте готовый файл SimpleCanvas.xaml в редакторе Xaml. Определение Canvas должно иметь следующий вид (в случае загрузки примеров в приложение WPF дескриптор Page нужно будет заменить дескриптором Window):

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="285" Width="325">
    <Canvas Background="LightSteelBlue">
```

```

<Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203"
        Width="80" Content="OK"/>
<Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
        Width="328" Height="25" FontSize="15"
        Content="Enter Car Information"/>
<Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60"
        Content="Make"/>
<TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60"
        Width="193" Height="25"/>
<Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109"
        Content="Color"/>
<TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107"
        Width="193" Height="25"/>
<Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155"
        Content="Pet Name"/>
<TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153"
        Width="193" Height="25"/>
</Canvas>
</Page>

```

В верхней половине экрана отобразится окно, показанное на рис. 25.1.

Обратите внимание, что порядок объявления элементов содержимого внутри Canvas не влияет на расчет местоположения; на самом деле местоположение основано на размере элемента управления и значениях его свойств Canvas.Top, Canvas.Bottom, Canvas.Left и Canvas.Right.

На заметку! Если подэлементы внутри Canvas не определяют специфическое местоположение с использованием синтаксиса присоединяемых свойств (например, Canvas.Left и Canvas.Top), тогда они автоматически прикрепляются к левому верхнему углу Canvas.

Применение типа Canvas может показаться предпочтительным способом организации содержимого (т.к. он выглядит настолько знакомым), но данному подходу присущи некоторые ограничения. Во-первых, элементы внутри Canvas не изменяют свои размеры динамически при использовании стилей или шаблонов (скажем, их шрифты остаются незатронутыми). Во-вторых, панель Canvas не пытается сохранять элементы видимыми, когда конечный пользователь уменьшает размер окна.

Пожалуй, наилучшим применением типа Canvas является позиционирование *графического содержимого*. Например, при построении изображения с использованием XAML определенно понадобится сделать так, чтобы все линии, фигуры и текст оставались на своих местах, а не динамически перемещались в случае изменения пользователем размера окна. Мы еще вернемся к Canvas в главе 26 при обсуждении служб визуализации графики WPF.



Рис. 25.1. Диспетчер компоновки Canvas делает возможным абсолютное позиционирование содержимого

Позиционирование содержимого внутри панелей WrapPanel

Панель WrapPanel позволяет определять содержимое, которое будет протекать сквозь панель, когда размер окна изменяется. При позиционировании элементов внутри WrapPanel их координаты верхнего левого и правого нижнего углов не указываются, как обычно делается в Canvas. Однако для каждого подэлемента допускается определение значений свойств Height и Width (наряду с другими свойствами), чтобы управлять их общим размером в контейнере.

Поскольку содержимое внутри WrapPanel не пристыковывается к заданной стороне панели, порядок объявления элементов играет важную роль (содержимое визуализируется от первого элемента до последнего). В файле SimpleWrapPanel.xaml находится следующая разметка (заключенная внутрь определения Page):

```
<WrapPanel Background="LightSteelBlue">
    <Label x:Name="lblInstruction" Width="328"
        Height="25" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Content="Make"/>
    <TextBox x:Name="txtMake" Width="193" Height="25"/>
    <Label x:Name="lblColor" Content="Color"/>
    <TextBox x:Name="txtColor" Width="193" Height="25"/>
    <Label x:Name="lblPetName" Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Width="193" Height="25"/>
    <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

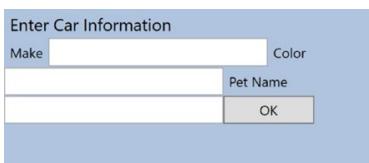


Рис. 25.2. Содержимое в панели WrapPanel ведет себя во многом подобно традиционной странице HTML

Когда эта разметка загружена, при изменении ширины окна содержимое выглядит не особо привлекательно, т.к. оно перетекает слева направо внутри окна (рис. 25.2).

По умолчанию содержимое WrapPanel перетекает слева направо. Тем не менее, если изменить значение свойства Orientation на Vertical, то можно заставить содержимое перетекать сверху вниз:

```
<WrapPanel Background="LightSteelBlue"
    Orientation="Vertical">
```

Панель WrapPanel (как и ряд других типов панелей) может быть объявлена с указанием значений ItemWidth и ItemHeight, которые управляют стандартным размером каждого элемента. Если подэлемент предоставляет собственные значения Height и/или Width, то он будет позиционироваться относительно размера, установленного для него панелью. Взгляните на следующую разметку:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="100" Width="650">
    <WrapPanel Background="LightSteelBlue"
        Orientation ="Horizontal" ItemWidth ="200" ItemHeight ="30">
        <Label x:Name="lblInstruction" FontSize="15"
            Content="Enter Car Information"/>
        <Label x:Name="lblMake" Content="Make"/>
        <TextBox x:Name="txtMake"/>
```

```

<Label x:Name="lblColor" Content="Color"/>
<TextBox x:Name="txtColor"/>
<Label x:Name="lblPetName" Content="Pet Name"/>
<TextBox x:Name="txtPetName"/>
<Button x:Name="btnOK" Width ="80" Content="OK"/>
</WrapPanel>
</Page>

```

В результате визуализации получается окно, показанное на рис. 25.3 (обратите внимание на размер и позицию элемента управления Button, для которого было задано уникальное значение Width).



Рис. 25.3. Панель WrapPanel может устанавливать ширину и высоту отдельного элемента

После просмотра рис. 25.3 вы наверняка согласитесь с тем, что панель WrapPanel — обычно не лучший выбор для организации содержимого непосредственно в окне, поскольку ее элементы могут беспорядочно смешиваться, когда пользователь изменяет размер окна. В большинстве случаев WrapPanel будет подэлементом панели другого типа, позволяя небольшой области окна переносить свое содержимое при изменении размера (как, например, элемент управленияToolBar).

Позиционирование содержимого внутри панелей StackPanel

Подобно WrapPanel элемент управления StackPanel организует содержимое внутри одиночной строки, которая может быть ориентирована горизонтально или вертикально (по умолчанию) в зависимости от значения, присвоенного свойству Orientation. Однако отличие между ними заключается в том, что StackPanel не пытается переносить содержимое при изменении размера окна пользователем. Взамен элементы в StackPanel просто растягиваются (согласно выбранной ориентации), приспособливаясь к размеру самой панели StackPanel. Например, в файле SimpleStackPanel.xaml содержится разметка, которая в результате дает вывод, показанный на рис. 25.4:

```

<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Panels!" Height="200" Width="400">
<StackPanel Background="LightSteelBlue" Orientation ="Vertical">
    <Label Name="lblInstruction"
        FontSize="15" Content="Enter Car Information"/>
    <Label Name="lblMake" Content="Make"/>
    <TextBox Name="txtMake"/>
    <Label Name="lblColor" Content="Color"/>
    <TextBox Name="txtColor"/>
    <Label Name="lblPetName" Content="Pet Name"/>
    <TextBox Name="txtPetName"/>
    <Button Name="btnOK" Width ="80" Content="OK"/>
</StackPanel>
</Page>

```

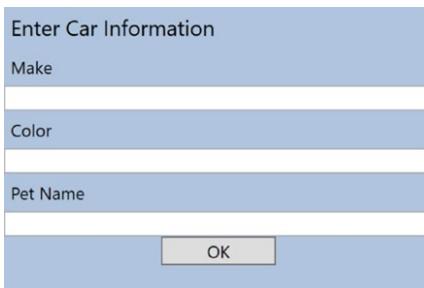


Рис. 25.4. Вертикальное укладывание содержимого

Если присвоить свойству `Orientation` значение `Horizontal`, тогда визуализированный вывод станет таким, как на рис. 25.5:

```
<StackPanel Background="LightSteelBlue" Orientation="Horizontal">
```



Рис. 25.5. Горизонтальное укладывание содержимого

Подобно `WrapPanel` панель `StackPanel` тоже редко применяется для организации содержимого прямо внутри окна. Панель `StackPanel` должна использоваться как вложенная панель в какой-нибудь главной панели.

Позиционирование содержимого внутри панелей Grid

Из всех панелей, предоставляемых API-интерфейсами WPF, панель `Grid` является, несомненно, самой гибкой. Аналогично таблице HTML панель `Grid` может состоять из набора ячеек, каждая из которых имеет свое содержимое. При определении `Grid` выполняются перечисленные ниже шаги.

1. Определение и конфигурирование каждой колонки.
2. Определение и конфигурирование каждой строки.
3. Назначение содержимого каждой ячейке сетки с применением синтаксиса присоединяемых свойств.

На заметку! Если не определить какие-либо строки и колонки, то по умолчанию элемент `Grid` будет состоять из единственной ячейки, которая заполняет всю поверхность окна. Кроме того, если не установить ячейку (колонку и строку) для подэлемента внутри `Grid`, тогда он автоматически разместится в колонке 0 и строке 0.

Первые два шага (определение колонок и строк) выполняются с использованием элементов `Grid.ColumnDefinitions` и `Grid.RowDefinitions`, которые содержат коллекции элементов `ColumnDefinition` и `RowDefinition` соответственно. Каждая ячейка внутри сетки на самом деле является подлинным объектом .NET, так что можно желаемым образом настраивать внешний вид и поведение каждого элемента.

Ниже представлено простое определение `Grid` (из файла `SimpleGrid.xaml`), которое организует содержимое пользовательского интерфейса, как показано на рис. 25.6:

```
<Grid ShowGridLines ="True" Background ="LightSteelBlue">
    <!-- Определить строки и колонки -->
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <!-- Добавить элементы в ячейки сетки -->
    <Label x:Name="lblInstruction" Grid.Column ="0" Grid.Row ="0"
        FontSize="15" Content="Enter Car Information"/>
    <Button x:Name="btnOK" Height ="30" Grid.Column ="0"
        Grid.Row ="0" Content="OK"/>
    <Label x:Name="lblMake" Grid.Column ="1"
        Grid.Row ="0" Content="Make"/>
    <TextBox x:Name="txtMake" Grid.Column ="1"
        Grid.Row ="0" Width="193" Height="25"/>
    <Label x:Name="lblColor" Grid.Column ="0"
        Grid.Row ="1" Content="Color"/>
    <TextBox x:Name="txtColor" Width="193" Height="25"
        Grid.Column ="0" Grid.Row ="1" />

    <!-- Добавить цвет к ячейке с именем, просто чтобы сделать
        картину интереснее -->
    <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
    <Label x:Name="lblPetName" Grid.Column ="1" Grid.Row ="1"
        Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Grid.Column ="1" Grid.Row ="1"
        Width="193" Height="25"/>
</Grid>
```

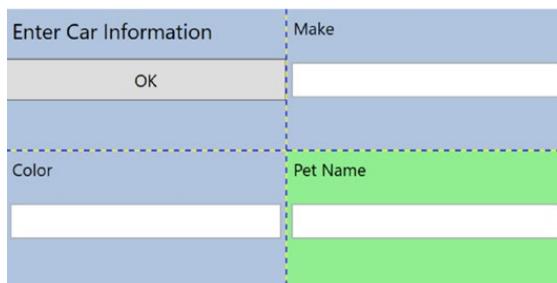


Рис. 25.6. Панель `Grid` в действии

Обратите внимание, что каждый элемент (включая элемент Rectangle светло-зеленого цвета) прикрепляется к ячейке сетки с применением присоединяемых свойств Grid.Row и Grid.Column. По умолчанию порядок ячеек начинается с левой верхней ячейки, которая указывается с использованием Grid.Column="0" и Grid.Row="0". Учитывая, что сетка состоит всего из четырех ячеек, правая нижняя ячейка может быть идентифицирована как Grid.Column="1" и Grid.Row="1".

Установка размеров столбцов и строк в панели Grid

Задавать размеры столбцов и строк в панели Grid можно одним из трех способов:

- установка абсолютных размеров (например, 100);
- установка автоматических размеров;
- установка относительных размеров (например, 3*).

Установка абсолютных размеров — именно то, что и можно было ожидать; для размера колонки (или строки) указывается специфическое число единиц, независимых от устройства. При установке автоматических размеров размер каждой колонки или строки определяется на основе элементов управления, содержащихся в колонке или строке. Установка относительных размеров практически эквивалентна заданию размеров в процентах внутри стиля CSS. Общая сумма чисел в колонках или строках с относительными размерами распределяется на общий объем доступного пространства.

В следующем примере первая строка получает 25% пространства, а вторая — 75% пространства:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="1*" />
  <ColumnDefinition Width="3*" />
</Grid.ColumnDefinitions>
```

Панели Grid с типами GridSplitter

Панели Grid также способны поддерживать разделители. Как вам возможно известно, разделители позволяют конечному пользователю изменять размеры колонок и строк сетки. При этом содержимое каждой ячейки с изменяемым размером реорганизует себя на основе находящихся в нем элементов. Добавлять разделители к Grid довольно просто: необходимо определить элемент управления GridSplitter и с применением синтаксиса присоединяемых свойств указать строку или колонку, на которую он воздействует.

Имейте в виду, что для того, чтобы разделитель был виден на экране, потребуется присвоить значение его свойству Width или Height (в зависимости от вертикального или горизонтального разделения). Ниже показана простая панель Grid с разделителем на первой колонке (Grid.Column="0") из файла GridWithSplitter.xaml:

```
<Grid Background ="LightSteelBlue">
  <!-- Определить колонки -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width ="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
```

```

<!-- Добавить метку в ячейку 0 -->
<Label x:Name="lblLeft" Background ="GreenYellow"
       Grid.Column ="0" Content ="Left!" />
<!-- Определить разделитель -->
<GridSplitter Grid.Column ="0" Width ="5"/>
<!-- Добавить метку в ячейку 1 -->
<Label x:Name="lblRight" Grid.Column ="1" Content ="Right!" />
</Grid>

```

Прежде всего, обратите внимание, что колонка, которая будет поддерживать разделитель, имеет свойство `Width`, установленное в `Auto`. В добавок элемент `GridSplitter` использует синтаксис присоединяемых свойств для указания, с какой колонкой он работает. В выводе (рис. 25.7) можно заметить 5-пиксельный разделитель, который позволяет изменять размер каждого элемента `Label`. Из-за того, что для элементов `Label` не было задано свойство `Height` или `Width`, они заполняют всю ячейку.

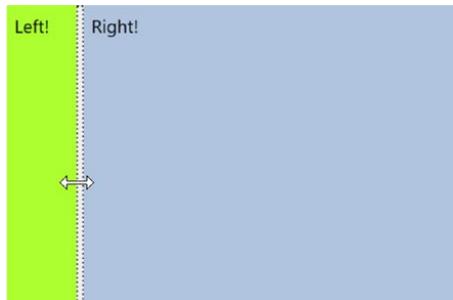


Рис. 25.7. Панель Grid с разделителем

Позиционирование содержимого внутри панелей DockPanel

Панель `DockPanel` обычно применяется в качестве контейнера, который содержит любое количество дополнительных панелей для группирования связанного содержимого. Панели `DockPanel` используют синтаксис присоединяемых свойств (как было показано в типах `Canvas` и `Grid`) для управления местом, куда будет пристыковываться каждый элемент внутри `DockPanel`.

В файле `SimpleDockPanel.xaml` определена следующая простая панель `DockPanel`, которая дает результат, показанный на рис. 25.8:

```

<DockPanel LastChildFill ="True" Background="AliceBlue">
    <!-- Стыковать элементы к панели -->
    <Label DockPanel.Dock ="Top" Name="lblInstruction" FontSize="15"
          Content="Enter Car Information"/>
    <Label DockPanel.Dock ="Left" Name="lblMake" Content="Make"/>
    <Label DockPanel.Dock ="Right" Name="lblColor" Content="Color"/>
    <Label DockPanel.Dock ="Bottom" Name="lblPetName" Content="Pet Name"/>
    <Button Name="btnOK" Content="OK"/>
</DockPanel>

```

На заметку! Если добавить множество элементов к одной стороне `DockPanel`, то они выстроются вдоль указанной грани в порядке их объявления.



Рис. 25.8. Простая панель DockPanel

Преимущество применения типов DockPanel заключается в том, что при изменении пользователем размера окна каждый элемент остается прикрепленным к указанной (посредством DockPanel.Dock) стороне панели. Также обратите внимание, что внутри открывающего дескриптора DockPanel в этом примере атрибут LastChildFill установлен в true. Поскольку элемент Button на самом деле является “последним дочерним” элементом в контейнере, он будет растянут, чтобы занять все оставшееся пространство.

Включение прокрутки в типах панелей

Полезно упомянуть, что в рамках инфраструктуры WPF поставляется класс ScrollViewer, который обеспечивает автоматическое поведение прокрутки данных внутри объектов панелей. Вот как он определяется в файле SimpleScrollViewer.xaml:

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="50"/>
    <Button Content ="Second" Background = "Red" Height ="50"/>
    <Button Content ="Third" Background = "Pink" Height ="50"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="50"/>
    <Button Content ="Fifth" Background = "Blue" Height ="50"/>
  </StackPanel>
</ScrollViewer>
```

Результат визуализации приведенного определения XAML представлен на рис. 25.9 (обратите внимание на то, что справа в окне отображается линейка прокрутки, т.к. размера окна не хватает, чтобы показать все пять кнопок).



Рис. 25.9. Работа с классом ScrollViewer

Как и можно было ожидать, каждый класс панели предлагает многочисленные члены, позволяющие точно настраивать размещение содержимого. В качестве связанныго замечания: многие элементы управления WPF поддерживают два удобных свойства (*Padding* и *Margin*), которые предоставляют элементу управления возможность самостоятельного информирования панели о том, как с ним следует обращаться. В частности, свойство *Padding* управляет тем, сколько свободного пространства должно окружать внутренний элемент управления, а свойство *Margin* контролирует объем дополнительного пространства вне элемента управления.

На этом краткий экскурс в основные типы панелей WPF и различные способы позиционирования их содержимого завершен. Далее будет показано, как использовать визуальные конструкторы Visual Studio для создания компоновок.

Конфигурирование панелей с использованием визуальных конструкторов Visual Studio

Теперь, когда вы ознакомились с разметкой XAML, применяемой при определении ряда общих диспетчеров компоновки, полезно знать, что IDE-среда Visual Studio предлагает очень хорошую поддержку для конструирования компоновок. Ключевым компонентом является окно *Document Outline*, описанное ранее в главе. Чтобы проиллюстрировать некоторые основы, мы создадим новый проект приложения WPF по имени *VisualLayoutTester*.

В первоначальной разметке для *Window* по умолчанию используется диспетчер компоновки *Grid*:

```
<Window x:Class="VisualLayoutTester.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:VisualLayoutTesterApp"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
    <Grid>
    </Grid>
</Window>
```

Если вы благополучно применяете систему компоновки *Grid*, то на рис. 25.10 заметите, что можно легко разделять и менять размеры ячеек сетки, используя визуальный конструктор. Сначала необходимо выбрать компонент *Grid* в окне *Document Outline* и затем щелкнуть на границе сетки, чтобы создать новые строки и колонки.

Теперь предположим, что определена сетка с каким-то числом ячеек. Далее можно перетаскивать элементы управления в интересующую ячейку сетки и IDE-среда будет автоматически устанавливать их свойства *Grid.Row* и *Grid.Column*. Вот как может выглядеть разметка, сгенерированная IDE-средой после перетаскивания элемента *Button* в предопределенную ячейку:

```
<Button x:Name="button" Content="Button" Grid.Column="1"
    HorizontalAlignment="Left"
    Margin="21,21.4,0,0" Grid.Row="1"
    VerticalAlignment="Top" Width="75"/>
```

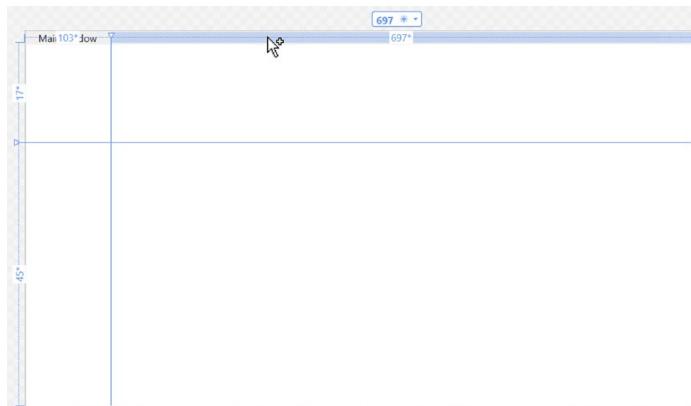


Рис. 25.10. Элемент управления *Grid* может быть разделен на ячейки с применением визуального конструктора IDE-среды

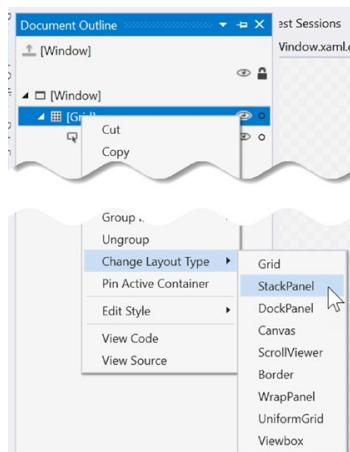


Рис. 25.11. Окно Document Outline позволяет выполнять преобразования в другие типы панелей

Пусть, например, было решено вообще не использовать элемент *Grid*. Щелчок правой кнопкой мыши на любом узле разметки в окне Document Outline приводит к открытию контекстного меню, которое содержит пункт, позволяющий заменить текущий контейнер другим (рис. 25.11). Следует осознавать, что такое действие (с высокой вероятностью) радикально изменит позиции элементов управления, потому что они станут удовлетворять правилам нового типа панели.

Еще один удобный трюк связан с возможностью выбора в визуальном конструкторе набора элементов управления и последующего их группирования внутри нового вложенного диспетчера компоновки. Предположим, что имеется панель *Grid*, которая содержит набор произвольных объектов. Выделите множество элементов на поверхности визуального конструктора, щелкнув на каждом элементе левой кнопкой мыши при нажатой клавише <Ctrl>. Если вы затем щелкните правой кнопкой мыши на выбранном наборе, то с помощью открывшегося контекстного меню сможете сгруппировать выделенные элементы в новую вложенную панель (рис. 25.12).

После этого снова загляните в окно Document Outline, чтобы проконтролировать вложенную систему компоновки. Так как строятся полнофункциональные окна WPF, скорее всего, всегда нужно будет использовать систему вложенных компоновок, а не просто выбирать единственную панель для отображения всего пользовательского интерфейса (фактически в оставшихся примерах приложений WPF обычно так и будет делаться). В качестве финального замечания следует указать, что все узлы в окне Document Outline поддерживают перетаскивание.

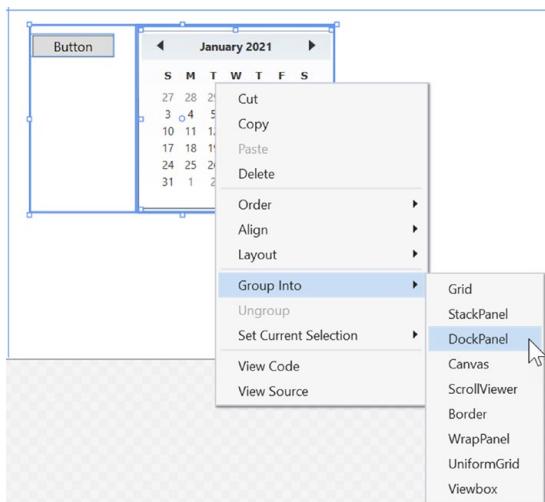


Рис. 25.12. Группирование элементов в новую вложенную панель

Например, если требуется переместить в родительскую панель управления, который в текущий момент находится внутри Canvas, тогда можно поступить так, как иллюстрируется на рис. 25.13.

В последующих главах, посвященных WPF, будут представлены дополнительные ускоренные приемы для работы с компоновкой там, где они возможны. Тем не менее, определенно полезно посвятить какое-то время самостоятельному экспериментированию и проверке разнообразных средств. В следующем примере данной главы будет демонстрироваться построение вложенного диспетчера компоновки для специального приложения обработки текста (с проверкой правописания).

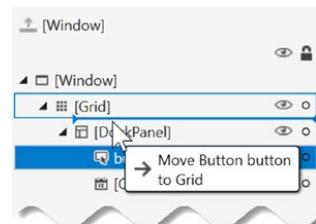


Рис. 25.13. Перемещение элементов с помощью окна Document Outline

Построение окна с использованием вложенных панелей

Как упоминалось ранее, в типичном окне WPF для получения желаемой системы компоновки применяется не единственный элемент управления типа панели, а одни панели вкладываются внутрь других. Начните с создания нового проекта приложения WPF по имени MyWordPad.

Вашей целью является конструирование компоновки, в которой главное окно имеет расположенную в верхней части систему меню, под ней — панель инструментов и в нижней части окна — строку состояния. Страна состояния будет содержать область для текстовых подсказок, которые отображаются при выборе пользователем пункта меню (или кнопки в панели инструментов). Система меню и панель инструментов предоставят триггеры пользовательского интерфейса для закрытия приложения и отображения вариантов правописания в виджете Expander.

На рис. 25.14 показана начальная компоновка; она также иллюстрирует возможности правописания в рамках инфраструктуры WPF.

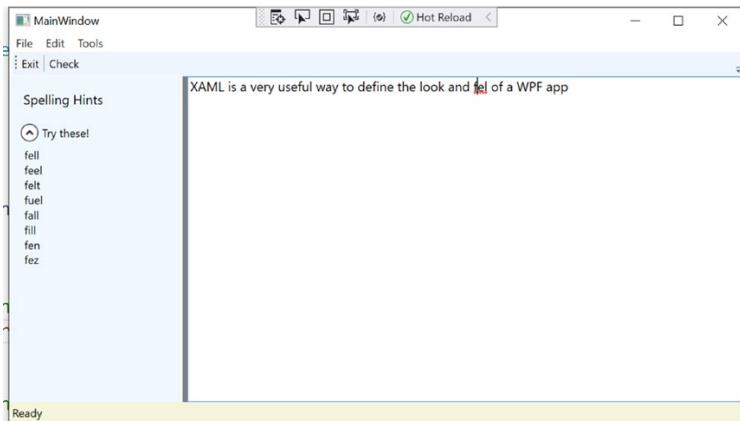


Рис. 25.14. Использование вложенных панелей для формирования пользовательского интерфейса окна

Чтобы приступить к построению интересующего пользовательского интерфейса, модифицируйте начальное определение XAML типа Window для использования дочернего элемента DockPanel вместо стандартного элемента управления Grid:

```
<Window x:Class="MyWordPad.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:MyWordPad"
    mc:Ignorable="d"
    Title="My Spell Checker" Height="450" Width="800">
    <!-- Эта панель устанавливает содержимое окна -->
    <DockPanel>
    </DockPanel>
</Window>
```

Построение системы меню

Системы меню в WPF представлены классом `MenuItem`, который поддерживает коллекцию объектов `MenuItem`. При построении системы меню в XAML каждый объект `MenuItem` можно заставить обрабатывать разнообразные события, наиболее примечательным из которых является `Click`, возникающее при выборе подэлемента конечным пользователем. В рассматриваемом примере создаются два пункта меню верхнего уровня (`File` (Файл) и `Tools` (Сервис)); позже будет построено меню `Edit` (Правка), которые содержат в себе подэлементы `Exit` (Выход) и `Spelling Hints` (Подсказки по правописанию) соответственно.

В дополнение к обработке события `Click` для каждого подэлемента необходимо также обработать события `MouseEnter` и `MouseExit`, которые применяются для установки текста в строке состояния. Добавьте в контекст элемента `DockPanel` следующую разметку:

```
<!-- Стыковать систему меню к верхней части -->
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left" Background="White" BorderBrush ="Black">
    <MenuItem Header="_File">
      <Separator/>
      <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
                MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
                MouseEnter ="MouseEnterToolsHintsArea"
                MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"/>
    </MenuItem>
  </Menu>
```

Обратите внимание, что система меню стыкована с верхней частью DockPanel. Кроме того, элемент Separator применяется для добавления в систему меню тонкой горизонтальной линии прямо перед пунктом Exit. Значения Header для каждого MenuItem содержат символ подчеркивания (например, _Exit). Подобным образом указывается символ, который будет подчеркиваться, когда конечный пользователь нажмет клавишу <Alt> (для ввода клавиатурного сокращения). Символ подчеркивания используется вместо символа & в Windows Forms, т.к. язык XAML основан на XML, а символ & в XML имеет особый смысл.

После построения системы меню необходимо реализовать различные обработчики событий. Прежде всего, есть обработчик пункта меню `File⇒Exit` (Файл⇒Выход). `FileExit_Click()`, который просто закрывает окно, что в свою очередь приводит к завершению приложения, поскольку это окно самого высшего уровня. Обработчики событий `MouseEnter` и `MouseExit` для каждого подэлемента будут в итоге обновлять строку состояния; однако пока просто оставьте их пустыми. Наконец, обработчик `ToolsSpellingHints_Click()` для пункта меню `Tools⇒Spelling Hints` также оставьте пока пустым. Ниже показаны текущие обновления файла отделенного кода (в том числе обновленные операторы `using`):

```
protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
{
}
protected void MouseEnterToolsHintsArea(object sender,
                                         RoutedEventArgs args)
{
}
protected void MouseLeaveArea(object sender, RoutedEventArgs args)
{
}
}
```

Визуальное построение меню

Наряду с тем, что всегда полезно знать, как вручную определять элементы в XAML, такая работа может быть слегка утомительной. В Visual Studio поддерживается возможность визуального конструирования систем меню, панелей инструментов, строк состояния и многих других элементов управления пользовательского интерфейса. Щелчок правой кнопкой мыши на элементе управления `Menu` приводит к открытию контекстного меню, содержащего `Add MenuItem` (Добавить MenuItem), который позволяет добавить новый пункт меню в элемент управления `Menu`. После добавления набора пунктов верхнего уровня можно заняться добавлением пунктов подменю, разделителей, развертыванием и свертыванием самого меню и выполнением других связанных с меню операций посредством второго щелчка правой кнопкой мыши.

В оставшейся части примера MyWordPad вы увидите финальную сгенерированную разметку XAML; тем не менее, посвятите некоторое время экспериментированию с визуальными конструкторами.

Построение панели инструментов

Панели инструментов (представляемые в WPF классом `ToolBar`) обычно предлагаются альтернативный способ активизации пунктов меню. Поместите следующую разметку непосредственно после закрывающего дескриптора определения `Menu`:

```
<!-- Поместить панель инструментов под областью меню -->
<ToolBar DockPanel.Dock ="Top" >
    <Button Content ="Exit" MouseEnter ="MouseEnterExitArea"
           MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    <Separator/>
    <Button Content ="Check" MouseEnter ="MouseEnterToolsHintsArea"
           MouseLeave ="MouseLeaveArea"
           Click ="ToolsSpellingHints_Click"
           Cursor="Help" />
</ToolBar>
```

Ваш элемент управления ToolBar образован из двух элементов управления Button, которые предназначены для обработки тех же самых событий теми же методами из файла кода. С помощью такого приема можно дублировать обработчики для обслуживания и пунктов меню, и кнопок панели инструментов. Хотя в данной панели применяются типичные нажимаемые кнопки, вы должны принимать во внимание, что тип ToolBar "является" ContentControl, а потому на его поверхность можно помещать любые типы (скажем, раскрывающиеся списки, изображения и графику). Еще один интересный аспект связан с тем, что кнопка Check (Проверить) поддерживает специальный курсор мыши через свойство Cursor.

На заметку! Элемент ToolBar может быть дополнительно помещен внутрь элемента ToolBarTray, который управляет компоновкой, стыковкой и перетаскиванием для набора объектов ToolBar.

Построение строки состояния

Элемент управления строкой состояния (StatusBar) стыкуется с нижней частью DockPanel и содержит единственный элемент управления TextBlock, который ранее в главе не использовался. Элемент TextBlock можно применять для хранения текста с форматированием вроде выделения полужирным и подчеркивания, добавления разрывов строк и т.д. Поместите приведенную ниже разметку сразу после предыдущего определения элемента управления ToolBar:

```
<!-- Разместить строку состояния внизу -->
<StatusBar DockPanel.Dock ="Bottom".Background="Beige" >
    <StatusBarItem>
        <TextBlock Name="statBarText" Text="Ready"/>
    </StatusBarItem>
</StatusBar>
```

Завершение проектирования пользовательского интерфейса

Финальный аспект проектирования нашего пользовательского интерфейса связан с определением поддерживающего разделители элемента Grid, в котором определены две колонки. Слева находится элемент управления Expander, помещенный внутрь StackPanel, который будет отображать список предполагаемых вариантов правописания, а справа — элемент TextBox с поддержкой многострочного текста, линеек прокрутки и включенной проверкой орфографии. Элемент Grid может быть целиком размещен в левой части родительской панели DockPanel. Чтобы завершить определение пользовательского интерфейса окна, добавьте следующую разметку XAML, расположив ее непосредственно под разметкой, которая описывает StatusBar:

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
    <!-- Определить строки и колонки -->
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
    <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
        <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
            Spelling Hints
        </Label>

        <Expander Name="expanderSpelling" Header ="Try these!" Margin="10,10,10,10">
            <!-- Будет заполняться программно -->
            <Label Name ="lblSpellingHints" FontSize ="12"/>
        </Expander>
    </StackPanel>

    <!-- Это будет областью для ввода -->
    <TextBox Grid.Column ="1" >
```

```

        SpellCheck.IsEnabled = "True"
        AcceptsReturn = "True"
        Name = "txtData" FontSize = "14"
        BorderBrush = "Blue"
        VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto">
    </TextBox>
</Grid>
```

Реализация обработчиков событий MouseEnter/MouseLeave

К настоящему моменту пользовательский интерфейс окна готов. Понадобится лишь предоставить реализации оставшихся обработчиков событий. Начните с обновления файла кода C# так, чтобы каждый из обработчиков событий MouseEnter и MouseLeave устанавливал в текстовой панели строки состояния подходящее сообщение, которое окажет помощь конечному пользователю:

```

public partial class MainWindow : System.Windows.Window
{
    ...
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Exit the Application";
    }
    protected void MouseEnterToolsHintsArea(object sender,
                                             RoutedEventArgs args)
    {
        statBarText.Text = "Show Spelling Suggestions";
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Ready";
    }
}
```

Теперь приложение можно запустить. Текст в строке состояния должен изменяться в зависимости от того, над каким пунктом меню или кнопкой панели инструментов находится курсор.

Реализация логики проверки правописания

Инфраструктура WPF имеет встроенную поддержку проверки правописания, независимую от продуктов Microsoft Office. Это значит, что использовать уровень взаимодействия с COM для обращения к функции проверки правописания Microsoft Word не понадобится; та же самая функциональность добавляется с помощью всего нескольких строк кода.

Вспомните, что при определении элемента управления TextBox свойство SpellCheck.IsEnabled устанавливается в true. В результате неправильно написанные слова подчеркиваются красной волнистой линией, как происходит в Microsoft Office. Более того, лежащая в основе программная модель предоставляет доступ к механизму проверки правописания, который позволяет получить список предполагаемых вариантов для слов, написанных с ошибкой.

Добавьте в метод ToolsSpellingHints_Click() следующий код:

```

protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
    string spellingHints = string.Empty;
    // Попробовать получить ошибку правописания
    // в текущем положении курсора ввода.
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
        // Построить строку с предполагаемыми вариантами правописания.
        foreach (string s in error.Suggestions)
        {
            spellingHints += $"{s}\n";
        }
        // Отобразить предполагаемые варианты и раскрыть элемент Expander.
        lblSpellingHints.Content = spellingHints;
        expanderSpelling.IsExpanded = true;
    }
}

```

Приведенный выше код довольно прост. С применением свойства `CaretIndex` извлекается объект `SpellingError` и вычисляется текущее положение курсора ввода в текстовом поле. Если в указанном месте присутствует ошибка (т.е. значение `error` не равно `null`), тогда осуществляется проход в цикле по списку предполагаемых вариантов с использованием свойства `Suggestions`. После того, как все предполагаемые варианты для неправильно написанного слова получены, они помещаются в элемент `Label` внутри элемента `Expander`.

Вот и все! С помощью нескольких строк процедурного кода (и приличной порции разметки XAML) заложены основы для функционирования текстового процессора. После изучения [управляющих команд](#) будут добавлены дополнительные возможности.

Понятие команд WPF

Инфраструктура WPF предлагает поддержку того, что может считаться [независимыми от элементов управления событиями](#), через архитектуру команд. Обычное событие .NET Core определяется внутри некоторого базового класса и может использоваться только этим классом или его потомками. Следовательно, нормальные события .NET Core тесно привязаны к классу, в котором они определены.

По контрасту команды WPF представляют собой похожие на события сущности, которые не зависят от специфического элемента управления и во многих случаях могут успешно применяться к многочисленным (и на вид несвязанным) типам элементов управления. Вот лишь несколько примеров: WPF поддерживает команды копирования, вырезания и вставки, которые могут использоваться в разнообразных элементах пользовательского интерфейса (вроде пунктов меню, кнопок панели инструментов и специальных кнопок), а также клавиатурные комбинации (скажем, `<Ctrl+C>` и `<Ctrl+V>`).

В то время как другие инструментальные наборы, ориентированные на построение пользовательских интерфейсов (вроде Windows Forms), предлагают для таких целей стандартные события, их применение обычно дает в результате избыточный и трудный в сопровождении код. Внутри модели WPF в качестве альтернативы можно использовать команды. Итогом обычно оказывается более компактная и гибкая кодовая база.

Внутренние объекты команд

Инфраструктура WPF поставляется с множеством встроенных команд, каждую из которых можно ассоциировать с соответствующей клавиатурной комбинацией (или другим входным жестом). С точки зрения программирования команда WPF — это любой объект, поддерживающий свойство (часто называемое *Command*), которое возвращает объект, реализующий показанный ниже интерфейс *ICommand*:

```
public interface ICommand
{
    // Возникает, когда происходят изменения, влияющие
    // на то, должна выполняться команда или нет.
    event EventHandler CanExecuteChanged;

    // Определяет метод, который выясняет, может ли
    // команда выполняться в ее текущем состоянии.
    bool CanExecute(object parameter);

    // Определяет метод для вызова при обращении к команде.
    void Execute(object parameter);
}
```

В WPF предлагаются разнообразные классы команд, которые открывают доступ к примерно сотне готовых объектов команд. В таких классах определены многочисленные свойства, представляющие специфические объекты команд, каждый из которых реализует интерфейс *ICommand*. В табл. 25.3 кратко описаны избранные стандартные объекты команд.

Таблица 25.3. Внутренние объекты команд WPF

Класс WPF	Объекты команд	Описание
ApplicationCommands	Close, Copy, Cut, Delete, Find, Open, Paste, Save, SaveAs, Redo, Undo	Разнообразные команды уровня приложения
ComponentCommands	MoveDown, MoveFocusBack, MoveLeft, MoveRight, ScrollToEnd, ScrollToHome	Разнообразные команды, которые являются общими для компонентов пользовательского интерфейса
MediaCommands	BoostBase, ChannelUp, ChannelDown, FastForward, NextTrack, Play, Rewind, Select, Stop	Разнообразные команды, связанные с мультимедиа
NavigationCommands	BrowseBack, BrowseForward, Favorites, LastPage, NextPage, Zoom	Разнообразные команды, связанные с навигационной моделью WPF
EditingCommands	AlignCenter, CorrectSpellingError, DecreaseFontSize, EnterLineBreak, EnterParagraphBreak, MoveDownByLine, MoveRightByWord	Разнообразные команды, связанные с интерфейсом Documents API в WPF

Подключение команд к свойству Command

Для подключения любого свойства команд WPF к элементу пользовательского интерфейса, который поддерживает свойство Command (такому как Button или MenuItem), потребуется проделать совсем небольшую работу. В качестве примера модифицируйте текущую систему меню, добавив новый пункт верхнего уровня по имени Edit (Правка) с тремя подэлементами, которые позволяют копировать, вставлять и вырезать текстовые данные:

```
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left"
      Background="White" BorderBrush ="Black">
    <MenuItem Header="_File" Click ="FileExit_Click" >
      <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
                MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    </MenuItem>
    <!-- Новые пункты меню с командами -->
    <MenuItem Header="_Edit">
      <MenuItem Command ="ApplicationCommands.Copy"/>
      <MenuItem Command ="ApplicationCommands.Cut"/>
      <MenuItem Command ="ApplicationCommands.Paste"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
                MouseEnter ="MouseEnterToolsHintsArea"
                MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"/>
    </MenuItem>
</Menu>
```

Обратите внимание, что свойству Command каждого подэлемента в меню Edit присвоено некоторое значение. В результате пункты меню автоматически получают корректные имена и горячие клавиши (например, <Ctrl+C> для операции вырезания) в пользовательском интерфейсе меню, и приложение теперь способно копировать, вырезать и вставлять текст без необходимости в написании процедурного кода.

Если вы запустите приложение и выделите какую-то часть текста, то сразу же сможете пользоваться новыми пунктами меню. Вдобавок приложение также оснащено возможностью реагирования на стандартную операцию щелчка правой кнопкой мыши, предлагая пользователю те же самые пункты в контекстном меню.

Подключение команд к произвольным действиям

Если объект команды нужно подключить к произвольному событию (специфично для приложения), то придется прибегнуть к написанию процедурного кода. Задача несложная, но требует чуть больше логики, чем можно видеть в XAML. Например, пусть необходимо, чтобы все окно реагировало на нажатие клавиши <F1>, активизируя ассоциированную с ним справочную систему. Также предположим, что в файле кода для главного окна определен новый метод по имени SetF1CommandBinding(), который вызывается внутри конструктора после вызова InitializeComponent():

```
public MainWindow()
{
  InitializeComponent();
  SetF1CommandBinding();
}
```

Метод SetF1CommandBinding() будет программно создавать новый объект CommandBinding, который можно применять всякий раз, когда требуется привязать объект команды к заданному обработчику событий в приложении. Сконфигурируйте объект CommandBinding для работы с командой ApplicationCommands.Help, которая автоматически выдается по нажатию клавиши <F1>:

```
private void SetF1CommandBinding()
{
    CommandBinding helpBinding =
        new CommandBinding(ApplicationCommands.Help);
    helpBinding.CanExecute += CanHelpExecute;
    helpBinding.Executed += HelpExecuted;
    CommandBindings.Add(helpBinding);
}
```

Большинство объектов CommandBinding будет обрабатывать событие CanExecute (которое позволяет указать, инициируется ли команда для конкретной операции программы) и событие Executed (где можно определить код, подлежащий выполнению после того, как команда произошла). Добавьте к типу, производному от Window, следующие обработчики событий (форматы методов регламентируются ассоциированными делегатами):

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
    // Если нужно предотвратить выполнение команды,
    // то можно установить CanExecute в false.
    e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Look, it is not that difficult. Just type something!",
                    "Help!");
}
```

В предыдущем фрагменте кода метод CanHelpExecute() реализован так, что справка по нажатию <F1> всегда разрешена; это делается путем возвращения true. Однако если в определенных ситуациях справочная система отображаться не должна, то необходимо предпринять соответствующую проверку и возвращать false. Созданная "справочная система", отображаемая внутри HelpExecute(), представляет собой всего лишь обычное окно сообщения. Теперь можете запустить приложение. После нажатия <F1> появится ваше окно сообщения.

Работа с командами Open и Save

Чтобы завершить текущий пример, вы добавите функциональность сохранения текстовых данных во внешнем файле и открытия файлов *.txt для редактирования. Можно пойти длинным путем, вручную добавив программную логику, которая включает и отключает пункты меню в зависимости от того, имеются ли данные внутри TextBox. Тем не менее, для сокращения усилий можно прибегнуть к услугам команд.

Начните с обновления элемента MenuItem, который представляет меню File верхнего уровня, путем добавления двух новых подменю, использующих объекты Save и Open класса ApplicationCommands:

```
<MenuItem Header="_File">
    <MenuItem Command ="ApplicationCommands.Open"/>
    <MenuItem Command ="ApplicationCommands.Save"/>
    <Separator/>
    <MenuItem Header ="_Exit"
        MouseEnter ="MouseEnterExitArea"
        MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
</MenuItem>
```

Вспомните, что все объекты команд реализуют интерфейс `ICommand`, в котором определены два события (`CanExecute` и `Executed`). Теперь необходимо разрешить окну выполнять указанные команды, предварительно проверив возможность делать это в текущих обстоятельствах; раз так, можете определить обработчик события для запуска специального кода.

Понадобится наполнить коллекцию `CommandBindings`, поддерживаемую окном. В разметке XAML потребуется применить синтаксис “свойство-элемент” для определения области `Window.CommandBindings`, в которую помещаются два определения `CommandBinding`. Модифицируйте определение `Window`, как показано ниже:

```
<Window x:Class="MyWordPad.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MySpellChecker" Height="331" Width="508"
    WindowStartupLocation ="CenterScreen" >
    <!-- Это информирует элемент управления Window о том, какие
        обработчики вызывать при поступлении команд Open и Save -->
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Open"
            Executed="OpenCmdExecuted"
            CanExecute="OpenCmdCanExecute"/>
        <CommandBinding Command="ApplicationCommands.Save"
            Executed="SaveCmdExecuted"
            CanExecute="SaveCmdCanExecute"/>
    </Window.CommandBindings>
    <!-- Эта панель устанавливает содержимое окна -->
    <DockPanel>
        ...
    </DockPanel>
</Window>
```

Щелкните правой кнопкой мыши на каждом из атрибутов `Executed` и `CanExecute` в редакторе XAML и выберите в контекстном меню пункт `Navigate to Event Handler` (Перейти к обработчику события). Как объяснялось в главе 24, в результате автоматически генерируется заготовка кода для обработчика события. Теперь в файле кода C# для окна должны присутствовать четыре пустых обработчика событий.

Реализация обработчиков события `CanExecute` будет сообщать окну, что можно инициировать соответствующие события `Executed` в любой момент, для чего свойство `CanExecute` входного объекта `CanExecuteRoutedEventArgs` устанавливается в `true`:

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

```
private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

Обработчики соответствующего события Executed выполняют действительную работу по отображению диалоговых окон открытия и сохранения файла; они также отправляют данные из TextBox в файл. Начните с импортирования пространств имен System.IO и Microsoft.Win32 в файл кода. Окончательный код прямолинеен:

```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    // Создать диалоговое окно открытия файла и показать
    // в нем только текстовые файлы.
    var openDlg = new OpenFileDialog { Filter = "Text Files (*.txt)" };
    // Был ли совершен щелчок на кнопке OK?
    if (true == openDlg.ShowDialog())
    {
        // Загрузить содержимое выбранного файла.
        string dataFromFile = File.ReadAllText(openDlg.FileName);
        // Отобразить строку в TextBox.
        txtData.Text = dataFromFile;
    }
}

private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
    var saveDlg = new SaveFileDialog { Filter = "Text Files (*.txt)" };
    // Был ли совершен щелчок на кнопке OK?
    if (true == saveDlg.ShowDialog())
    {
        // Сохранить данные из TextBox в указанном файле.
        File.WriteAllText(saveDlg.FileName, txtData.Text);
    }
}
```

На заметку! Система команд WPF более подробно рассматривается в главе 28, где будут создаваться специальные команды на основе ICommand и RelayCommand.

Итак, пример и начальное знакомство с элементами управления WPF завершены. Вы узнали, как работать с базовыми командами, системами меню, строками состояния, панелями инструментов,ложенными панелями и несколькими основными элементами пользовательского интерфейса (вроде TextBox и Expander). В следующем примере вы будете иметь дело с более экзотическими элементами управления, а также с рядом важных служб WPF.

Понятие маршрутизуемых событий

Вы могли заметить, что в предыдущем примере кода передавался параметр RoutedEventArgs, а не EventArgs. Модель маршрутизуемых событий является усовершенствованием стандартной модели событий CLR и спроектирована для того, чтобы обеспечить возможность обработки событий в манере, подходящей описанию

XAML дерева объектов. Предположим, что имеется новый проект приложения WPF по имени `WpfRoutedEvents`. Модифицируйте описание XAML начального окна, добавив следующий элемент управления `Button`, который определяет сложное содержимое:

```
<Window ...
  <Grid>
    <Button Name="btnClickMe" Height="75" Width = "250"
      Click ="btnClickMe_Clicked">
      <StackPanel Orientation ="Horizontal">
        <Label Height="50" FontSize ="20">Fancy Button!</Label>
        <Canvas Height ="50" Width ="100" >
          <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
            Width ="50" Cursor="Hand" Canvas.Left="25"
            Canvas.Top="12"/>
          <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15"
            Width ="36" Canvas.Top="17" Canvas.Left="32"/>
        </Canvas>
      </StackPanel>
    </Button>
  </Grid>
</Window>
```

Обратите внимание, что в открывавшем определении элемента `Button` было обработано событие `Click` за счет указания имени метода, который должен вызываться при возникновении события. Событие `Click` работает с делегатом `RoutedEventHandler`, который ожидает обработчик события, принимающий `object` в первом параметре и `System.Windows.RoutedEventArgs` во втором. Реализуйте такой обработчик:

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
  // Делать что-нибудь, когда на кнопке произведен щелчок.
  MessageBox.Show("Clicked the button");
}
```

После запуска приложения окно сообщения будет отображаться независимо от того, на какой части содержимого кнопки был выполнен щелчок (зеленый элемент `Ellipse`, желтый элемент `Ellipse`, элемент `Label` или поверхность элемента `Button`). В принципе это хорошо. Только представьте, насколько громоздким оказалась бы обработка событий WPF, если бы пришлось обрабатывать событие `Click` для каждого из упомянутых подэлементов. Дело не только в том, что создание отдельных обработчиков событий для каждого аспекта `Button` — трудоемкая задача, а еще и в том, что в результате получился бы сложный в сопровождении код.

К счастью, маршрутизуемые события WPF позаботятся об автоматическом вызове единственного обработчика события `Click` вне зависимости от того, на какой части кнопки был совершен щелчок. Выражаясь просто, модель маршрутизуемых событий автоматически распространяет событие вверх (или вниз) по дереву объектов в поисках подходящего обработчика.

Точнее говоря, маршрутазиуемое событие может использовать три стратегии маршрутизации. Если событие перемещается от точки возникновения вверх к другим областям определений внутри дерева объектов, то его называют *пузырьковым событием*. И наоборот, если событие перемещается от самого внешнего элемента (например, `Window`) вниз к точке возникновения, то его называют *туннельным событием*. Наконец, если событие инициируется и обрабатывается только элементом, внутри ко-

торого оно возникло (что можно было бы описать как нормальное событие CLR), то его называют *прямым событием*.

Роль пузырьковых маршрутизируемых событий

В текущем примере, когда пользователь щелкает на внутреннем овале желто-го цвета, событие Click поднимается на следующий уровень области определения (Canvas), затем на StackPanel и в итоге на уровень Button, где обрабатывается. Подобным же образом, если пользователь щелкает на Label, то событие всплывает на уровень StackPanel и, в конце концов, попадает в элемент Button.

Благодаря такому шаблону пузырьковых маршрутизируемых событий не придется беспокоиться о регистрации специфичных обработчиков события Click для всех членов составного элемента управления. Однако если необходимо выполнить специальную логику обработки щелчков для нескольких элементов внутри того же самого дерева объектов, то это вполне можно делать.

В целях иллюстрации предположим, что щелчок на элементе управления outerEllipse должен быть обработан в уникальной манере. Сначала обработайте событие MouseDown для этого подэлемента (графически визуализируемые типы вроде Ellipse не поддерживают событие Click, но могут отслеживать действия кнопки мыши через события MouseDown, MouseUp и т.д.):

```
<Button Name="btnClickMe" Height="75" Width = "250"
    Click ="btnClickMe_Clicked">
    <StackPanel Orientation ="Horizontal">
        <Label Height="50" FontSize ="20">Fancy Button!</Label>
        <Canvas Height ="50" Width ="100" >
            <Ellipse Name = "outerEllipse" Fill ="Green"
                Height ="25" MouseDown ="outerEllipse_MouseDown"
                Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
            <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
                Canvas.Top="17" Canvas.Left="32"/>
        </Canvas>
    </StackPanel>
</Button>
```

Затем реализуйте подходящий обработчик событий, который в демонстрационных целях будет просто изменять свойство Title главного окна:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";
}
```

Далее можно выполнять разные действия в зависимости от того, на чем конкретно щелкнул конечный пользователь (на внешнем эллипсе или в любом другом месте внутри области кнопки).

На заметку! Пузырьковые маршрутизируемые события всегда перемещаются из точки возникновения до следующей определяющей области. Таким образом, в рассмотренном примере щелчок на элементе innerEllipse привел бы к попаданию события в контейнер Canvas, а не в элемент outerEllipse, потому что оба элемента являются типами Ellipse внутри области определения Canvas.

Продолжение или прекращение пузырькового распространения

В текущий момент, когда пользователь щелкает на объекте outerEllipse, запускается зарегистрированный обработчик события MouseDown для данного объекта Ellipse, после чего событие всплывает до события Click кнопки. Чтобы информировать WPF о необходимости останова пузырькового распространения по дереву объектов, свойство Handled параметра MouseButtonEventArgs понадобится установить в true:

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить заголовок окна.
    this.Title = "You clicked the outer ellipse!";

    // Остановить пузырьковое распространение.
    e.Handled = true;
}
```

В таком случае обнаружится, что заголовок окна изменился, но окно MessageBox, отображаемое обработчиком события Click элемента Button, не появляется. По существу пузырьковые маршрутизируемые события позволяют сложной группе содер-жимого действовать либо как единый логический элемент (например, Button), либо как отдельные элементы (скажем, Ellipse внутри Button).

Роль туннельных маршрутизируемых событий

Строго говоря, маршрутизируемые события по своей природе могут быть пузырьковыми (как было описано только что) или туннельными. Туннельные события (имена которых начинаются с префикса Preview — наподобие PreviewMouseDown) спускаются от самого верхнего элемента до внутренних областей определения дерева объектов. В общем и целом для каждого пузырькового события в библиотеках базовых классов WPF предусмотрено связанное туннельное событие, которое возникает перед его пузырьковым аналогом. Например, перед возникновением пузырькового события MouseDown сначала инициируется туннельное событие PreviewMouseDown.

Обработка туннельных событий выглядит очень похожей на обработку любых других событий: нужно просто указать имя обработчика события в разметке XAML (или при необходимости применить соответствующий синтаксис обработки событий C# в файле кода) и реализовать такой обработчик в коде. Для демонстрации взаимодействия туннельных и пузырьковых событий начните с организации обработки события PreviewMouseDown для объекта outerEllipse:

```
<Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
        MouseDown ="outerEllipse_MouseDown"
        PreviewMouseDown ="outerEllipse_PreviewMouseDown"
        Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

Затем модифицируйте текущее определение класса C#, обновив обработчики событий (для всех объектов) за счет добавления данных о событии в переменную-член _mouseActivity типа string с использованием входного объекта аргументов события. В результате появится возможность наблюдать за потоком событий, появляю-щихся в фоновом режиме.

334 Часть VIII. Разработка клиентских приложений для Windows

```
public partial class MainWindow : Window
{
    string _mouseActivity = string.Empty;
    public MainWindow()
    {
        InitializeComponent();
    }

    public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
    {
        AddEventInfo(sender, e);
        MessageBox.Show(_mouseActivity, "Your Event Info");
        // Очистить строку для следующего цикла.
        _mouseActivity = "";
    }

    private void AddEventInfo(object sender, RoutedEventArgs e)
    {
        _mouseActivity += string.Format(
            "{0} sent a {1} event named {2}.\\n",
            sender,
            e.RoutedEventArgs.RoutingStrategy,
            e.RoutedEventArgs.Name);
    }

    private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
    {
        AddEventInfo(sender, e);
    }

    private void outerEllipse_PreviewMouseDown(object sender,
                                                MouseButtonEventArgs e)
    {
        AddEventInfo(sender, e);
    }
}
```

Обратите внимание, что ни в одном обработчике событий пузырьковое распространение не останавливается. После запуска приложения отобразится окно с уникальным сообщением, которое зависит от места на кнопке, где был произведен щелчок. На рис. 25.15 показан результат щелчка на внешнем объекте Ellipse.

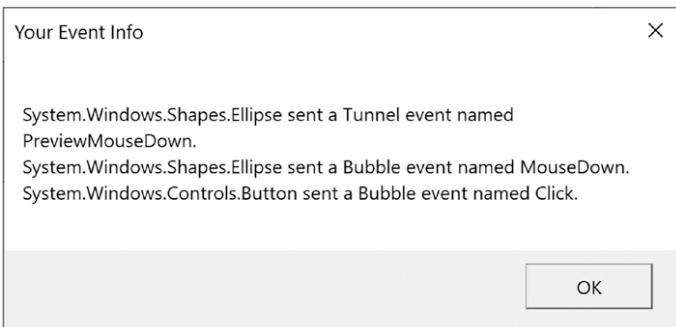


Рис. 25.15. Туннельное событие возникает первым, пузырьковое – вторым

Итак, почему события WPF обычно встречаются парами (одно туннельное и одно пузырьковое)? Ответ можно сформулировать так: благодаря предварительному просмотру событий появляется возможность выполнения любой специальной логики (проверки достоверности данных, отключения пузырькового распространения и т.п.) перед запуском пузырькового аналога событий. В качестве примера предположим, что создается элемент TextBox, который должен содержать только числовые данные. В нем можно было бы обработать событие PreviewKeyDown; если выясняется, что пользователь ввел нечисловые данные, то пузырьковое событие легко отменить, установив свойство Handled в true.

Как несложно было предположить, при построении специального элемента управления, который поддерживает специальные события, событие допускается реализовать так, чтобы оно могло распространяться пузырьковым (или туннельным) образом по дереву разметки XAML. В настоящей главе мы не рассматриваем процесс создания специальных маршрутизируемых событий (хотя он не особо отличается от построения специального свойства зависимости). Если интересно, загляните в раздел "Routed Events Overview" ("Обзор маршрутизируемых событий") документации по .NET Core, где предлагается несколько обучающих руководств, которые помогут в освоении этой темы.

Более глубокое исследование API-интерфейсов и элементов управления WPF

В оставшемся материале главы будет построено новое приложение WPF с применением Visual Studio. Целью является создание пользовательского интерфейса, который состоит из виджета TabControl, содержащего набор вкладок. Каждая вкладка будет иллюстрировать несколько новых элементов управления WPF и интересные API-интерфейсы, которые могут быть задействованы в разрабатываемых проектах. Попутно вы также узнаете о дополнительных возможностях визуальных конструкторов WPF из Visual Studio.

Работа с элементом управления TabControl

Первым делом создайте новый проект приложения WPF по имени WpfControlsAndAPIs. Как упоминалось ранее, начальное окно будет содержать элемент управления TabControl с четырьмя вкладками, каждая из которых отображает набор связанных элементов управления и/или API-интерфейсов WPF. Установите свойство Width окна в 800, а свойство Height окна в 350.

Перетащите элемент управления TabControl из панели инструментов Visual Studio на поверхность визуального конструктора и модифицируйте его разметку следующим образом:

```
<TabControl Name="MyTabControl" HorizontalAlignment="Stretch"
           VerticalAlignment="Stretch">
    <TabItem Header="TabItem">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="TabItem">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
</TabControl>
```

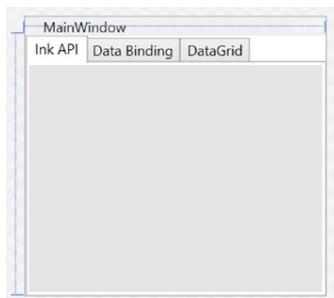


Рис. 25.16. Начальная компоновка системы вкладок

Вы заметите, что два элемента типа вкладок предоставляются автоматически. Чтобы добавить дополнительные вкладки, нужно щелкнуть правой кнопкой мыши на узле TabControl в окне Document Outline и выбрать в контекстном меню пункт Add TabItem (Добавить TabItem). Можно также щелкнуть правой кнопкой мыши на элементе TabControl в визуальном конструкторе и выбрать тот же самый пункт меню или просто ввести разметку в редакторе XAML. Добавьте одну дополнительную вкладку, используя любой из подходов.

Обновите разметку каждого элемента управления TabItem в редакторе XAML и измените их свойство Header, указывая Ink API, Data Binding и DataGrid. Окно визуального конструктора должно выглядеть примерно так, как на рис. 25.16.

Имейте в виду, что выбранная для редактирования вкладка становится активной, и можно формировать ее содержимое, перетаскивая элементы управления из панели инструментов. Располагая определением основного элемента управления TabControl, можно проработать детали каждой вкладки, одновременно изучая дополнительные средства API-интерфейса WPF.

Построение вкладки Ink API

Первая вкладка предназначена для раскрытия общей роли интерфейса Ink API, который позволяет легко встраивать в программу функциональность рисования. Конечно, его применение не ограничивается приложениями для рисования; Ink API можно использовать для разнообразных целей, включая фиксацию рукописного ввода.

На заметку! В оставшейся части главы (и в последующих главах, посвященных WPF) вместо применения разнообразных окон визуального конструктора будет главным образом на-прямую редактироваться разметка XAML. Хотя процедура перетаскивания элементов управления работает нормально, чаще всего компоновка оказывается нежелательной (Visual Studio добавляет границы и заполнение на основе того, где размещен элемент), а потому приходится тратить значительное время на очистку разметки XAML.

Начните с замены дескриптора Grid в элементе управления TabItem, помеченном как Ink API, дескриптором StackPanel и добавления закрывающего дескриптора. Разметка должна иметь такой вид:

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">
  </StackPanel>
</TabItem>
```

Проектирование панели инструментов

Добавьте (используя редактор XAML) в StackPanel новый элемент управленияToolBar по имени InkToolbar со свойством Height, установленным в 60:

```
<ToolBar Name="InkToolbar" Height="60">
</ToolBar>
```

Добавьте в ToolBar три элемента управления RadioButton внутри панели WrapPanel и элемента управления Border:

```
<Border Margin="0,2,0,2.4" Width="280" VerticalAlignment="Center">
  <WrapPanel>
    <RadioButton x:Name="inkRadio" Margin="5,10"
      Content="Ink Mode!" IsChecked="True" />
    <RadioButton x:Name="eraseRadio" Margin="5,10" Content="Erase Mode!" />
    <RadioButton x:Name="selectRadio" Margin="5,10" Content="Select Mode!" />
  </WrapPanel>
</Border>
```

Когда элемент управления RadioButton помещается не внутрь родительской панели, он получает пользовательский интерфейс, идентичный пользовательскому интерфейсу элемента управления Button! Именно потому элементы управления RadioButton были упакованы в панель WrapPanel.

Далее добавьте элемент Separator и элемент ComboBox, свойство Width которого установлено в 175, а свойство Margin — в 10,0,0,0. Добавьте три дескриптора ComboBoxItem с содержимым Red, Green и Blue и сопроводите весь элемент управления ComboBox еще одним элементом Separator:

```
<Separator/>
<ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0">
  <ComboBoxItem Content="Red"/>
  <ComboBoxItem Content="Green"/>
  <ComboBoxItem Content="Blue"/>
</ComboBox>
<Separator/>
```

Элемент управления RadioButton

В данном примере необходимо, чтобы три добавленных элемента управления RadioButton были взаимно исключающими. В других инфраструктурах для построения графических пользовательских интерфейсов такие связанные элементы требуют помещения в одну групповую рамку. Поступать подобным образом в WPF нет нужды. Взамен элементам управления просто назначается то же самое *групповое имя*, что очень удобно, поскольку связанные элементы не обязаны физически находиться внутри одной области, а могут располагаться где угодно в окне.

Класс RadioButton имеет свойство IsChecked, значения которого переключаются между true и false, когда конечный пользователь щелкает на элементе пользовательского интерфейса. К тому же элемент управления RadioButton предоставляет два события (Checked и Unchecked), которые можно применять для перехвата такого изменения состояния.

Добавление кнопок сохранения, загрузки и удаления

Финальным элементом управления внутри ToolBar будет Grid, содержащий три элемента управления Button. Поместите после последнего элемента управления Separator следующую разметку:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
```

```

<ColumnDefinition Width="Auto"/>
</Grid.ColumnDefinitions>
<Button Grid.Column="0" x:Name="btnSave" Margin="10,10"
        Width="70" Content="Save Data"/>
<Button Grid.Column="1" x:Name="btnLoad" Margin="10,10"
        Width="70" Content="Load Data"/>
<Button Grid.Column="2" x:Name="btnClear" Margin="10,10"
        Width="70" Content="Clear"/>
</Grid>

```

Добавление элемента управления InkCanvas

Финальным элементом управления для TabControl является InkCanvas. Поместите показанную ниже разметку после закрывающего дескриптора ToolBar, но перед закрывающим дескриптором StackPanel:

```
<InkCanvas x:Name="MyInkCanvas" Background="#FFB6F4F1" />
```

Предварительный просмотр окна

Теперь все готово к тестированию программы, для чего понадобится нажать клавишу <F5>. Должны отобразиться три взаимно исключающих переключателя, раскрывающийся список с тремя элементами и три кнопки (рис. 25.17).

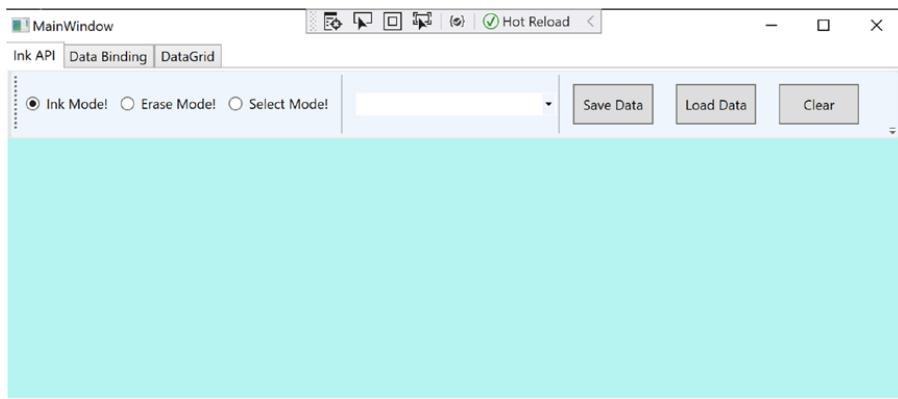


Рис. 25.17. Завершенная компоновка вкладки Ink API

Обработка событий для вкладки Ink API

Следующая задача для вкладки Ink API связана с организацией обработки события Click для каждого элемента управления RadioButton. Как вы поступали в других проектах WPF, просто щелкните на значке с изображением молнии в окне Properties среды Visual Studio и введите имена обработчиков событий. С помощью упомянутого приема свяжите событие Click каждого элемента управления RadioButton с тем же самым обработчиком по имени RadioButtonClicked. После обработки всех трех событий Click обработайте событие SelectionChanged элемента управления ComboBox, используя обработчик по имени ColorChanged. В результате должен получиться следующий код C#:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        this.InitializeComponent();
        // Вставить сюда код, требуемый при создании объекта.
    }
    private void RadioButtonClicked(object sender, RoutedEventArgs e)
    {
        // TODO: добавить сюда реализацию обработчика событий.
    }
    private void ColorChanged(object sender, SelectionChangedEventArgs e)
    {
        // TODO: добавить сюда реализацию обработчика событий.
    }
}

```

Обработчики событий будут реализованы позже, так что оставьте их пока пустыми.

Добавление элементов управления в панель инструментов

Вы добавите элемент управления InkCanvas путем прямого редактирования разметки XAML. Имейте в виду, что панель инструментов Visual Studio по умолчанию не отображает все возможные компоненты WPF, но содержимое панели инструментов можно обновлять.

Щелкните правой кнопкой мыши где-нибудь в области панели инструментов и выберите в контекстном меню пункт Choose Items (Выбрать элементы). Вскоре появится список возможных компонентов для добавления в панель инструментов. Вас интересует элемент управления InkCanvas (рис. 25.18).

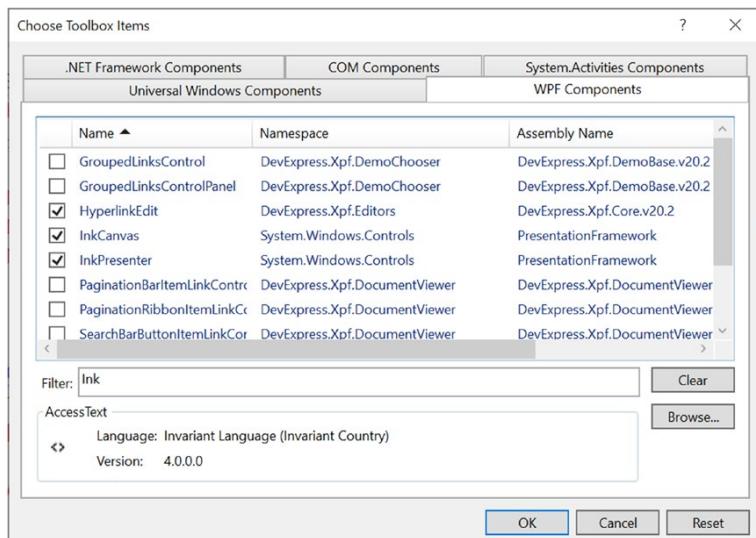


Рис. 25.18. Добавление новых компонентов в панель инструментов Visual Studio

На заметку! Элементы управления Ink API не совместимы с визуальным конструктором XAML в версии Visual Studio 16.8.3 (текущая версия на момент написания главы) или Visual Studio 16.9 Preview 2. Использовать элементы управления можно, но только не через визуальный конструктор.

Элемент управления InkCanvas

Простое добавление InkCanvas делает возможным рисование в окне. Рисовать можно с помощью мыши либо, если есть устройство, воспринимающее касания, то пальца или цифрового пера. Запустите приложение и нарисуйте что-нибудь (рис. 25.19).

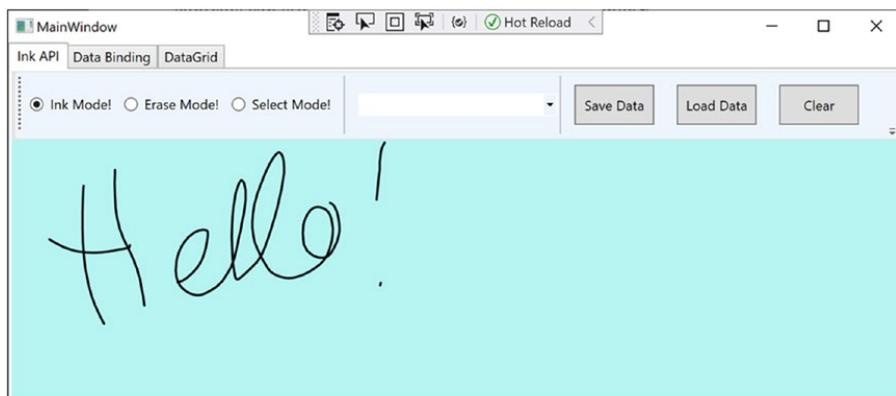


Рис. 25.19. Элемент управления InkCanvas в действии

Элемент управления InkCanvas обеспечивает нечто большее, чем просто рисование штрихов с помощью мыши (или пера); он также поддерживает несколько уникальных режимов редактирования, управляемых свойством `EditMode`, которому можно присвоить любое значение из связанного перечисления `InkCanvasEditMode`. В данном примере вас интересует режим `Ink`, принятый по умолчанию, который только что демонстрировался, режим `Select`, позволяющий пользователю выбирать с помощью мыши область для перемещения или изменения размера, и режим `EraseByStroke`, который удаляет предыдущий штрих мыши.

На заметку! Штрих — это визуализация, которая происходит во время одиночной операции нажатия и отпускания кнопки мыши. Элемент управления InkCanvas сохраняет все штрихи в объекте `StrokeCollection`, который доступен с применением свойства `Strokes`.

Обновите обработчик `RadioButtonClicked()` следующей логикой, которая помещает InkCanvas в нужный режим в зависимости от выбранного переключателя `RadioButton`:

```
private void RadioButtonClicked(object sender, RoutedEventArgs e)
{
    // В зависимости от того, какая кнопка отправила событие,
    // поместить InkCanvas в нужный режим оперирования.
```

```

this.MyInkCanvas.EditingMode
    = (sender as RadioButton)?.Content.ToString() switch
{
    // Эти строки должны совпадать со значениями свойства Content
    // каждого элемента RadioButton.
    "Ink Mode!" => InkCanvasEditingMode.Ink,
    "Erase Mode!" => InkCanvasEditingMode.EraseByStroke,
    "Select Mode!" => InkCanvasEditingMode.Select,
    _ => this.MyInkCanvas.EditingMode
};
}

```

В добавок установите Ink как стандартный режим в конструкторе окна. Там же установите стандартный выбор для ComboBox (элемент управления ComboBox более подробно рассматривается в следующем разделе):

```

public MainWindow()
{
    this.InitializeComponent();

    // Установить режим Ink в качестве стандартного.
    this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    this.inkRadio.IsChecked = true;
    this.comboColors.SelectedIndex = 0;
}

```

Теперь запустите программу еще раз, нажав <F5>. Войдите в режим Ink и нарисуйте что-нибудь. Затем перейдите в режим Erase и сотрите ранее нарисованное (курсор мыши автоматически примет вид стирающей резинки). Наконец, переключитесь в режим Select и выберите несколько линий, используя мышь в качестве лассо.

Охватив элемент, его можно перемещать по поверхности холста, а также изменять размеры. На рис. 25.20 демонстрируются разные режимы в действии.

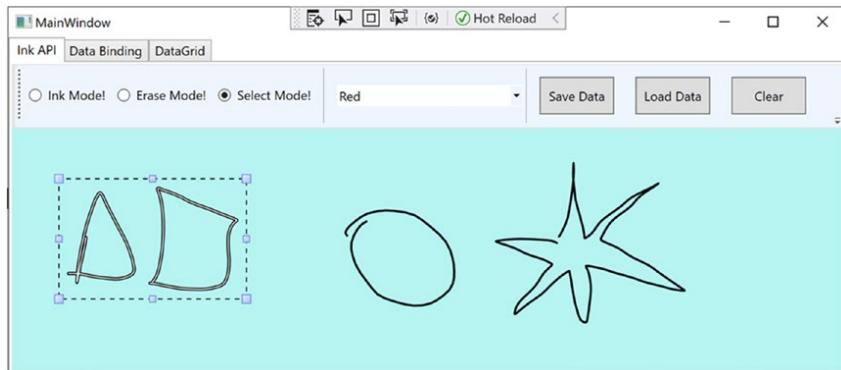


Рис. 25.20. Разные режимы редактирования элемента InkCanvas в действии

Элемент управления ComboBox

После заполнения элемента управления ComboBox (или ListBox) есть три способа определения выбранного в них элемента. Во-первых, когда необходимо найти числовой индекс выбранного элемента, должно применяться свойство SelectedIndex

(отсчет начинается с нуля; значение -1 представляет отсутствие выбора). Во-вторых, если требуется получить объект, выбранный внутри списка, то подойдет свойство SelectedItem. В-третьих, свойство SelectedValue позволяет получить значение выбранного объекта (обычно с помощью вызова ToString()).

Последний фрагмент кода, который понадобится добавить для данной вкладки, отвечает за изменение цвета штрихов, нарисованных в InkCanvas. Свойство DefaultDrawingAttributes элемента InkCanvas возвращает объект DrawingAttributes, который позволяет конфигурировать многочисленные аспекты пера, включая его размер и цвет (помимо других настроек). Модифицируйте код C# следующей реализацией метода ColorChanged():

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Получить выбранный элемент в раскрывающемся списке.
    string colorToUse =
        (this.comboColors.SelectedItem as ComboBoxItem)?.Content.ToString();

    // Изменить цвет, используемый для визуализации штрихов.
    this.MyInkCanvas.DefaultDrawingAttributes.Color =
        (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Вспомните, что ComboBox содержит коллекцию ComboBoxItems. В сгенерированной разметке XAML присутствует такое определение:

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
    <ComboBoxItem Content="Red"/>
    <ComboBoxItem Content="Green"/>
    <ComboBoxItem Content="Blue"/>
</ComboBox>
```

В результате обращения к свойству SelectedItem получается выбранный элемент ComboBoxItem, который хранится как экземпляр общего типа Object. После приведения Object к ComboBoxItem извлекается значение Content, которое будет строкой Red, Green или Blue. Эта строка затем преобразуется в объект Color с применением удобного служебного класса ColorConverter. Снова запустите программу. Теперь должна появиться возможность переключения между цветами при визуализации изображения.

Обратите внимание, что элементы управления ComboBox и ListBox также могут иметь сложное содержимое, а не только список текстовых данных. Чтобы получить представление о некоторых возможностях, откройте редактор XAML для окна и измените определение элемента управления ComboBox, поместив в него набор элементов StackPanel, каждый из которых содержит Ellipse и Label (свойство Width элемента ComboBox установлено в 175):

```
<ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0"
    SelectionChanged="ColorChanged">
    <StackPanel Orientation ="Horizontal" Tag="Red">
        <Ellipse Fill ="Red" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
            VerticalAlignment="Center" Content="Red"/>
    </StackPanel>
    <StackPanel Orientation ="Horizontal" Tag="Green">
        <Ellipse Fill ="Green" Height ="50" Width ="50"/>
```

```

<Label FontSize ="20" HorizontalAlignment="Center"
       VerticalAlignment="Center" Content="Green"/>
</StackPanel>
<StackPanel Orientation ="Horizontal" Tag="Blue">
    <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Blue"/>
</StackPanel>
</ComboBox>

```

В определении каждого элемента StackPanel выполняется присваивание значения свойству Tag, что является быстрым и удобным способом выявления, какой стек элементов был выбран пользователем (для этого существуют и лучшие способы, но пока достаточно такого). С указанной поправкой необходимо изменить реализацию метода ColorChanged():

```

private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
    // Получить свойство Tag выбранного элемента StackPanel.
    string colorToUse = (this.comboColors.SelectedItem
        as StackPanel).Tag.ToString();
    ...
}

```

После запуска программы элемент управления ComboBox будет выглядеть так, как показано на рис. 25.21.

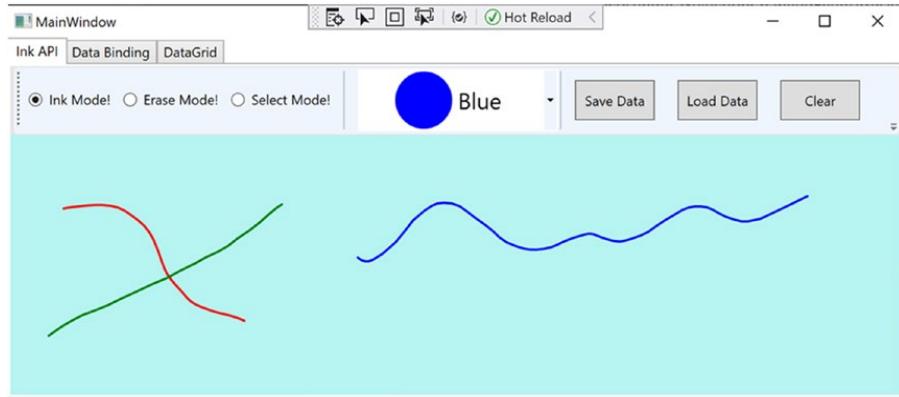


Рис. 25.21. Специальный элемент ComboBox, реализованный с помощью модели содержимого WPF

Сохранение, загрузка и очистка данных InkCanvas

Последняя часть вкладки Ink API позволит сохранять и загружать данные контейнера InkCanvas, а также очищать его содержимое, добавляя обработчики событий для кнопок в панели инструментов. Модифицируйте разметку XAML для кнопок за счет добавления разметки, отвечающей за события щелчков:

```
<Button Grid.Column="0" x:Name="btnSave" Margin="10,10"
        Width="70" Content="Save Data" Click="SaveData"/>
<Button Grid.Column="1" x:Name="btnLoad" Margin="10,10"
        Width="70" Content="Load Data" Click="LoadData"/>
<Button Grid.Column="2" x:Name="btnClear" Margin="10,10"
        Width="70" Content="Clear" Click="Clear"/>
```

Импортируйте пространства имен System.IO и System.Windows.Ink в файл кода. Реализуйте обработчики событий следующим образом:

```
private void SaveData(object sender, RoutedEventArgs e)
{
    // Сохранить все данные InkCanvas в локальном файле.
    using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
        this.MyInkCanvas.Strokes.Save(fs);
    fs.Close();
    MessageBox.Show("Image Saved", "Saved");
}
private void LoadData(object sender, RoutedEventArgs e)
{
    // Наполнить StrokeCollection из файла.
    using(FileStream fs = new FileStream("StrokeData.bin",
        FileMode.Open, FileAccess.Read))
        StrokeCollection strokes = new StrokeCollection(fs);
    this.MyInkCanvas.Strokes = strokes;
}
private void Clear(object sender, RoutedEventArgs e)
{
    // Очистить все штрихи.
    this.MyInkCanvas.Strokes.Clear();
}
```

Теперь должна появиться возможность сохранения данных в файле, их загрузки из файла и очистки InkCanvas от всех данных. Таким образом, работа с первой вкладкой элемента управления TabControl завершена, равно как и исследование интерфейса Ink API. Конечно, о технологии Ink API можно рассказать еще много чего, но теперь вы должны обладать достаточными знаниями, чтобы продолжить изучение темы самостоятельно. Далее вы узнаете, как применять привязку данных WPF.

Введение в модель привязки данных WPF

Элементы управления часто служат целью для разнообразных операций привязки данных. Выражаясь просто, привязка данных представляет собой действие по подключению свойств элемента управления к значениям данных, которые могут изменяться на протяжении жизненного цикла приложения. Это позволяет элементу пользовательского интерфейса отображать состояние переменной в коде. Например, привязку данных можно использовать для решения следующих задач:

- отмечать флажок элемента управления CheckBox на основе булевского свойства заданного объекта;
- отображать в элементах TextBox информацию, извлеченную из реляционной базы данных;
- подключать элемент Label к целому числу, представляющему количество файлов в папке.

При работе со встроенным механизмом привязки данных WPF важно помнить о разнице между *источником* и *местом назначения* операции привязки. Как и можно было ожидать, источником операции привязки данных являются сами данные (булевское свойство, реляционные данные и т.д.), а местом назначения (или целью) — свойство элемента управления пользовательского интерфейса, в котором задействуется содержимое данных (вроде свойства элемента управления CheckBox или TextBox).

В дополнение к привязке традиционных данных инфраструктура WPF делает возможной привязку элементов, как было продемонстрировано в предшествующих примерах. Это значит, что можно привязать (скажем) видимость свойства к свойству состояния отметки флажка. Такое действие было определено возможным в Windows Forms, но требовало реализации через код. Инфраструктура WPF предлагает развитую экосистему привязки данных, которая способна почти целиком поддерживаться в разметке. Она также позволяет обеспечивать синхронизацию источника и цели в случае изменения значений данных.

Построение вкладки Data Binding

В окне Document Outline замените элемент управления Grid во второй вкладке панелью StackPanel. Создайте следующую начальную компоновку с применением панели инструментов и окна Properties среды Visual Studio:

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
    <StackPanel Width="250">
        <Label Content="Move the scroll bar to see the current value"/>
        <!-- Значение линейки прокрутки является источником
            этой привязки данных -->
        <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
            Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>
        <!-- Содержимое метки будет привязано к линейке прокрутки -->
        <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
            BorderThickness="2" Content = "0"/>
    </StackPanel>
</TabItem>
```

Обратите внимание, что объект ScrollBar (названный здесь mySB) сконфигурирован с диапазоном от 1 до 100. Цель заключается в том, чтобы при изменении положения ползунка линейки прокрутки (либо по щелчку на символе стрелки влево или вправо) элемент Label автоматически обновлялся текущим значением. В настоящий момент значение свойства Content элемента управления Label установлено в "0"; тем не менее, оно будет изменено посредством операции привязки данных.

Установка привязки данных

Механизмом, обеспечивающим определение привязки в разметке XAML, является расширение разметки {Binding}. Хотя привязки можно определять посредством Visual Studio, это столь же легко делать прямо в разметке. Отредактируйте разметку XAML свойства Content элемента Label по имени labelSBThumb следующим образом:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
    BorderThickness="2" Content = "{Binding Value, ElementName=mySB}"/>
```

Обратите внимание на значение, присвоенное свойству Content элемента Label. Конструкция {Binding} обозначает операцию привязки данных. Значение ElementName представляет источник операции привязки данных (объект ScrollBar), а Path указывает свойство, к которому осуществляется привязка (свойство Value линейки прокрутки).

Если вы запустите программу снова, то обнаружите, что содержимое метки обновляется на основе значения линейки прокрутки по мере перемещения ползунка.

Свойство DataContext

Для определения операции привязки данных в XAML может использоваться альтернативный формат, при котором допускается разбивать значения, указанные расширением разметки {Binding}, за счет явной установки свойства DataContext в источник операции привязки:

```
<!-- Разбиение объекта и значения посредством DataContext -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2"
       DataContext = "{Binding ElementName=mySB}"
       Content = "{Binding Path=Value}" />
```

В текущем примере вывод будет идентичным. С учетом этого вполне вероятно вас интересует, в каких случаях необходимо устанавливать свойство DataContext явно. Поступать так может быть удобно из-за того, что подэлементы способны наследовать свои значения в дереве разметки.

Подобным образом можно легко устанавливать один и тот же источник данных для семейства элементов управления, не повторяя избыточные фрагменты XAML-разметки "{Binding ElementName=X, Path=Y}" во множестве элементов управления. Например, пусть в панель StackPanel вкладки добавлен новый элемент Button (вскоре вы увидите, почему он имеет настолько большой размер):

```
<Button Content="Click" Height="200"/>
```

Чтобы сгенерировать привязки данных для множества элементов управления, вы могли бы применить Visual Studio, но взамен введите модифицированную разметку в редакторе XAML:

```
<!-- Обратите внимание, что StackPanel устанавливает
      свойство DataContext -->
<StackPanel Background="#FFE5E5E5"
             DataContext = "{Binding ElementName=mySB}">
    ...
    <!-- Теперь оба элемента пользовательского интерфейса работают
        со значением линейки прокрутки уникальными путями -->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
          BorderThickness="2"
          Content = "{Binding Path=Value}" />
    <Button Content="Click" Height="200" FontSize = "{Binding Path=Value}"/>
</StackPanel>
```

Здесь свойство DataContext панели StackPanel устанавливается напрямую. Таким образом, при перемещении ползунка не только отображается текущее значение в элементе Label, но и в соответствии с этим текущим значением увеличивается размер шрифта элемента Button (на рис. 25.22 показан возможный вывод).

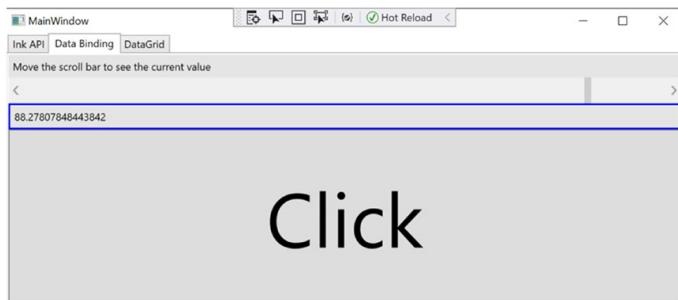


Рис. 25.22. Привязка значения ScrollBar к элементам Label и Button

Форматирование привязанных данных

Вместо ожидаемого целого числа для представления положения ползунка тип ScrollBar использует значение double. Следовательно, по мере перемещения ползунка внутри элемента Label будут отображаться разнообразные значения с плавающей точкой (вроде 61.0576923076923), которые выглядят не слишком интуитивно понятными для конечного пользователя, почти наверняка ожидающего увидеть целые числа (такие как 61, 62, 63 и т.д.).

При желании форматировать данные можно добавить свойство ContentStringFormat с передачей ему специальной строки и спецификатора формата .NET Core:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
    BorderThickness="2" Content = "{Binding Path=Value}"
    ContentStringFormat="The value is: {0:F0}"/>
```

Если в спецификаторе формата отсутствует какой-либо текст, тогда его понадобится предварить пустым набором фигурных скобок, который является управляющей последовательностью для XAML. Такой прием уведомляет процессор о том, что следующие за {} символы представляют собой литералы, а не, скажем, конструкцию привязки. Вот обновленная разметка XAML:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
    BorderThickness="2" Content = "{Binding Path=Value}"
    ContentStringFormat="{}{0:F0}"/>
```

На заметку! При привязке свойства Text элемента управления пару "имя-значение" объекта StringFormat можно добавлять прямо в конструкции привязки. Она должна быть отдельной только для свойств Content.

Преобразование данных с использованием интерфейса IValueConverter

Если требуется нечто большее, чем просто форматирование данных, тогда можно создать специальный класс, реализующий интерфейс IValueConverter из пространства имен System.Windows.Data. В интерфейсе IValueConverter определены два члена, позволяющие выполнять преобразование между источником и целью (в случае двунаправленной привязки). После определения такой класс можно применять для дальнейшего уточнения процесса привязки данных.

Вместо использования свойства форматирования можно применять преобразователь значений для отображения целых чисел внутри элемента управления Label. Добавьте в проект новый класс (по имени MyDoubleConverter) со следующим кодом:

```
using System;
using System.Windows.Data;
namespace WpfControlsAndAPIs
{
    public class MyDoubleConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            // Преобразовать значение double в int.
            double v = (double)value;
            return (int)v;
        }

        public object ConvertBack(object value, Type targetType,
object parameter, System.Globalization.CultureInfo culture)
        {
            // Поскольку заботиться здесь о "дву направленной" привязке
            // не нужно, просто возвратить значение value.
            return value;
        }
    }
}
```

Метод Convert() вызывается при передаче значения от источника (ScrollBar) к цели (свойство Content элемента Label). Хотя он принимает много входных аргументов, для такого преобразования понадобится манипулировать только входным аргументом типа object, который представляет текущее значение double. Данный тип можно использовать для приведения к целому и возврата нового числа.

Метод ConvertBack() будет вызываться, когда значение передается от цели к источнику (если включен двунаправленный режим привязки). Здесь мы просто возвращаем значение value. Это позволяет вводить в TextBox значение с плавающей точкой (например, 99.9) и автоматически преобразовывать его в целочисленное значение (99), когда пользователь перемещает фокус из элемента управления. Такое "бесплатное" преобразование происходит из-за того, что метод Convert() будет вызываться еще раз после вызова ConvertBack(). Если просто возвратить null из ConvertBack(), то синхронизация привязки будет выглядеть нарушенной, т.к. элемент TextBox по-прежнему будет отображать число с плавающей точкой.

Чтобы применить построенный преобразователь в разметке, сначала нужно создать локальный ресурс, представляющий только что законченный класс. Не переживайте по поводу механики добавления ресурсов; тема будет детально раскрыта в нескольких последующих главах. Поместите показанную ниже разметку сразу после открывающего дескриптора Window:

```
<Window.Resources>
<local:MyDoubleConverter x:Key="DoubleConverter"/>
</Window.Resources>
```

Далее обновите конструкцию привязки для элемента управления Label:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2"
Content = "{Binding Path=Value,Converter={StaticResource DoubleConverter}}"/>
```

Теперь после запуска приложения вы будете видеть только целые числа.

Установление привязок данных в коде

Специальный преобразователь данных можно также зарегистрировать в коде. Начните с очистки текущего определения элемента управления Label внутри вкладки Data Binding, чтобы расширение разметки {Binding} больше не использовалось:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" />
```

Добавьте оператор using для System.Windows.Data и в конструкторе окна вызовите новый закрытый вспомогательный метод по имени SetBindings(), код которого показан ниже:

```
using System.Windows.Data;
...
namespace WpfControlsAndAPIs
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            ...
            SetBindings();
        }
        ...
        private void SetBindings()
        {
            // Создать объект Binding.
            Binding b = new Binding
            {
                // Зарегистрировать преобразователь, источник и путь.
                Converter = new MyDoubleConverter(),
                Source = this.mySB,
                Path = new PropertyPath("Value")
                // Вызвать метод SetBinding() объекта Label.
                this.labelSBThumb.SetBinding(Label.ContentProperty, b);
            }
        }
    }
}
```

Единственная часть метода SetBindings(), которая может выглядеть несколько необычной — вызов SetBinding(). Обратите внимание, что первый параметр обращается к статическому, доступному только для чтения полю ContentProperty класса Label. Как вы узнаете далее в главе, такая конструкция называется *свойством зависимости*. Пока просто имейте в виду, что при установке привязки в коде первый аргумент почти всегда требует указания имени класса, нуждающегося в привяз-

ке (Label в рассматриваемом случае), за которым следует обращение к внутреннему свойству с добавлением к его имени суффикса `Property`. Запустив приложение, можно удостовериться в том, что элемент `Label` отображает только целые числа.

Построение вкладки `DataGrid`

В предыдущем примере привязки данных иллюстрировался способ конфигурирования двух (или большего количества) элементов управления для участия в операции привязки данных. Наряду с тем, что это удобно, возможно также привязывать данные из файлов XML, базы данных и объектов в памяти. Чтобы завершить текущий пример, вы должны спроектировать финальную вкладку элемента управления `DataGrid`, которая будет отображать информацию, извлеченную из таблицы `Inventory` базы данных `AutoLot`.

Как и с другими вкладками, начните с замены текущего элемента `Grid` панелью `StackPanel`, напрямую обновив разметку XAML в Visual Studio. Внутри нового элемента `StackPanel` определите элемент управления `DataGrid` по имени `gridInventory`:

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
    <StackPanel>
        <DataGrid x:Name="gridInventory" Height="288"/>
    </StackPanel>
</TabItem>
```

С помощью диспетчера пакетов NuGet добавьте в проект следующие пакеты:

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.Extensions.Configuration
Microsoft.Extensions.Configuration.Json
```

Если вы предпочитаете добавлять пакеты в интерфейсе командной строки .NET Core, тогда введите приведенные далее команды (в каталоге решения):

```
dotnet add WpfControlsAndAPIs package Microsoft.EntityFrameworkCore
dotnet add WpfControlsAndAPIs
    package Microsoft.EntityFrameworkCore.SqlServer
dotnet add WpfControlsAndAPIs package Microsoft.Extensions.Configuration
dotnet add WpfControlsAndAPIs
    package Microsoft.Extensions.Configuration.Json
```

Затем щелкните правой кнопкой мыши на имени решения, выберите в контекстном меню пункт `Add⇒Existing Project` (Добавить⇒Существующий проект) и добавьте проекты `AutoLot.Dal` и `AutoLot.Dal.Models` из главы 23, а также ссылки на эти проекты. Сделать это можно также с помощью интерфейса командной строки, выполнив показанные ниже команды (вам придется скорректировать пути к проектам согласно требованиям имеющейся операционной системы):

```
dotnet sln .\Chapter25_AllProjects.sln add ..\Chapter_23\AutoLot.Models
dotnet sln .\Chapter25_AllProjects.sln add ..\Chapter_23\AutoLot.Dal
dotnet add WpfControlsAndAPIs reference ..\Chapter_23\AutoLot.Models
dotnet add WpfControlsAndAPIs reference ..\Chapter_23\AutoLot.Dal
```

Убедитесь, что в проекте `AutoLot.Dal` все еще присутствует ссылка на проект `AutoLot.Dal.Models`. Добавьте в файл `MainWindow.xaml.cs` следующие пространства имен:

```
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Repos;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
```

Добавьте в MainWindow.cs два свойства уровня модуля для хранения экземпляров реализации IConfiguration и класса ApplicationContext:

```
private IConfiguration _configuration;
private ApplicationContext _context;
```

Добавьте новый метод по имени GetConfigurationAndDbContext() для хранения экземпляров реализации IConfiguration и класса ApplicationContext и вызовите его в конструкторе. Вот полный код метода:

```
private void GetConfigurationAndDbContext()
{
    _configuration = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", true, true)
        .Build();
    var optionsBuilder =
        new DbContextOptionsBuilder<ApplicationContext>();
    var connectionString =
        _configuration.GetConnectionString("AutoLot");
    optionsBuilder.UseSqlServer(connectionString,
        sqlOptions => sqlOptions.EnableRetryOnFailure());
    _context = new ApplicationContext(optionsBuilder.Options);
}
```

Добавьте в проект новый файл JSON по имени appsettings.json. Щелкните правой кнопкой мыши на имени этого файла в окне Solution Explorer, выберите в контекстном меню пункт Properties (Свойства) и установите свойство Copy To Output Directory (Копировать в выходной каталог) в Copy always (Всегда копировать). Вы можете добиться того же самого результата с помощью файла проекта:

```
<ItemGroup>
    <None Update="appsettings.json">
        <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </None>
</ItemGroup>
```

Модифицируйте файл JSON, как показано ниже (приведя строку подключения в соответствие со своей средой):

```
{
    "ConnectionStrings": {
        "AutoLotFinal": "server=.;5433;Database=AutoLot;
        User Id=sa;Password=P@ssw0rd;"}
}
```

Откройте файл MainWindow.xaml.cs, добавьте последнюю вспомогательную функцию по имени ConfigureGrid() и вызовите ее в конструкторе после конфигурирования ApplicationContext. Понадобится добавить лишь несколько строк кода:

```

private void ConfigureGrid()
{
    using var repo = new CarRepo(_context);
    gridInventory.ItemsSource = repo
        .GetAllIgnoreQueryFilters()
        .ToList()
        .Select(x=> new {
            x.Id,
            Make=x.MakeName,
            x.Color,
            x.PetName
        });
}

```

Запустив проект, вы увидите данные, заполняющие сетку. При желании сделать сетку более привлекательной можно применить окно Properties в Visual Studio для редактирования свойств сетки, чтобы улучшить ее внешний вид.

На этом текущий пример завершен. В последующих главах вы увидите в действии другие элементы управления, но к настоящему моменту вы должны чувствовать себя увереннее с процессом построения пользовательских интерфейсов в Visual Studio, а также при работе с разметкой XAML и кодом C#.

Роль свойств зависимости

Подобно любому API-интерфейсу .NET Core внутри WPF используется каждый член системы типов .NET Core (классы, структуры, интерфейсы, делегаты, перечисления) и каждый член типа (свойства, методы, события, константные данные, поля только для чтения и т.д.). Однако в WPF также поддерживается уникальная программная концепция под названием *свойство зависимости*.

Как и “нормальное” свойство .NET Core (которое в литературе, посвященной WPF, часто называют *свойством CLR*), свойство зависимости можно устанавливать декларативно с помощью разметки XAML или программно в файле кода. Кроме того, свойства зависимости (подобно свойствам CLR) в конечном итоге предназначены для инкапсуляции полей данных класса и могут быть сконфигурированы как доступные только для чтения, только для записи или для чтения и записи.

Вы будете практически всегда пребывать в блаженном неведении относительно того, что фактически устанавливаете (или читаете) свойство зависимости, а не свойство CLR! Например, свойства Height и Width, которые элементы управления WPF наследуют от класса FrameworkElement, а также член Content, унаследованный от класса ControlContent, на самом деле являются свойствами зависимости:

```

<!-- Установить три свойства зависимости -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100"
Content = "OK"/>

```

С учетом всех указанных сходств возникает вопрос: зачем нужно было определять в WPF новый термин для такой знакомой концепции? Ответ кроется в способе реализации свойства зависимости внутри класса. Пример кода будет показан позже, а на высоком уровне все свойства зависимости создаются описанным ниже способом.

- Класс, который определяет свойство зависимости, должен иметь в своей цепочке наследования класс DependencyObject.

- Одиночное свойство зависимости представляется как открытое, статическое, допускающее только чтение поле типа `DependencyProperty`. По соглашению это поле именуется путем снабжения имени оболочки CLR (см. последний пункт списка) суффиксом `Property`.
- Переменная типа `DependencyProperty` регистрируется посредством вызова статического метода `DependencyProperty.Register()`, который обычно происходит в статическом конструкторе или встраивается в объявление переменной.
- В классе будет определено дружественное к XAML свойство CLR, которое вызывает методы, предоставляемые классом `DependencyObject`, для получения и установки значения.

После реализации свойства зависимости предлагают несколько мощных инструментов, которые применяются разнообразными технологиями WPF, в том числе привязкой данных, службами анимации, стилями, шаблонами и т.д. Мотивацией создания свойств зависимости было желание предоставить способ вычисления значений свойств на основе значений из других источников. Далее приведен список основных преимуществ, которые выходят далеко за рамки простой инкапсуляции данных, обеспечиваемой свойствами CLR.

- Свойства зависимости могут наследовать свои значения от определения XAML родительского элемента. Например, если в открывающем дескрипторе `Window` определено значение для атрибута `FontSize`, то все элементы управления внутри `Window` по умолчанию будут иметь тот же самый размер шрифта.
- Свойства зависимости поддерживают возможность получать значения, которые установлены элементами внутри их области определения XAML, например, в случае установки элементом `Button` свойства `Dock` родительского контейнера `DockPanel`. (Вспомните, что присоединяемые свойства делают именно это, поскольку являются разновидностью свойств зависимости.)
- Свойства зависимости позволяют инфраструктуре WPF вычислять значение на основе множества внешних значений, что может быть важно для служб анимации и привязки данных.
- Свойства зависимости предоставляют поддержку инфраструктуры для триггеров WPF (также довольно часто используемых при работе с анимацией и привязкой данных).

Имейте в виду, что во многих случаях вы будете взаимодействовать с существующим свойством зависимости способом, идентичным работе с обычным свойством CLR (благодаря оболочке CLR). В предыдущем разделе, посвященном привязке данных, вы узнали, что если необходимо установить привязку данных в коде, то должен быть вызван метод `SetBinding()` на целевом объекте операции и указано свойство зависимости, с которым будет работать привязка:

```
private void SetBindings()
{
    Binding b = new Binding
    {
        // Зарегистрировать преобразователь, источник и путь.
        Converter = new MyDoubleConverter(),
        Source = this.mySB,
        Path = new PropertyPath("Value")
    };
}
```

354 Часть VIII. Разработка клиентских приложений для Windows

```
// Указать свойство зависимости.  
this.labelsSBThumb.SetBinding(Label.ContentProperty, b);  
}
```

Вы увидите похожий код в главе 27 во время исследования запуска анимации в коде:

```
// Указать свойство зависимости.  
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

Потребность в построении специального свойства зависимости возникает только во время разработки собственного элемента управления WPF. Например, когда создается класс `UserControl` с четырьмя специальными свойствами, которые должны тесно интегрироваться с API-интерфейсом WPF, они должны быть реализованы с применением логики свойств зависимости.

В частности, если нужно, чтобы свойство было целью операции привязки данных или анимации, если оно обязано уведомлять о своем изменении, если свойство должно быть в состоянии работать в качестве установщика в стиле WPF или получать свои значения от родительского элемента, то возможностей обычного свойства CLR окажется не достаточно. В случае использования обычного свойства другие программисты действительно могут получать и устанавливать его значение, но если они попытаются применить такое свойство внутри контекста службы WPF, то оно не будет работать ожидаемым образом. Поскольку заранее нельзя узнать, как другие пожелают взаимодействовать со свойствами специальных классов `UserControl`, нужно выработать в себе привычку при построении специальных элементов управления *всегда* определять свойства зависимости.

Исследование существующего свойства зависимости

Прежде чем вы научитесь создавать специальные свойства зависимости, давайте рассмотрим внутреннюю реализацию свойства `Height` класса `FrameworkElement`. Ниже приведен соответствующий код (с комментариями):

```
// FrameworkElement "является" DependencyObject.  
public class FrameworkElement : UIElement, IFrameworkInputElement,  
    IIInputElement, ISupportInitialize, IHaveResources, IQueryAmbient  
{  
    ...  
    // Статическое поле только для чтения типа DependencyProperty.  
    public static readonly DependencyProperty HeightProperty;  
    // Поле DependencyProperty часто регистрируется  
    // в статическом конструкторе класса.  
    static FrameworkElement()  
    {  
        ...  
        HeightProperty = DependencyProperty.Register(  
            "Height",  
            typeof(double),  
            typeof(FrameworkElement),  
            new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,  
                FrameworkPropertyMetadataOptions.AffectsMeasure,  
                new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),  
                new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));  
    }  
}
```

```
// Оболочка CLR, реализованная с использованием
// унаследованных методов GetValue() / SetValue() .
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```

Как видите, по сравнению с обычными свойствами CLR свойства зависимости требуют немалого объема дополнительного кода. В реальности зависимость может оказаться даже еще более сложной, чем показано здесь (к счастью, многие реализации проще свойства Height).

В первую очередь вспомните, что если в классе необходимо определить свойство зависимости, то он должен иметь в своей цепочке наследования DependencyObject, т.к. именно этот класс определяет методы GetValue() и SetValue(), применяемые в оболочке CLR. Из-за того, что класс FrameworkElement “является” DependencyObject, указанное требование удовлетворено.

Далее вспомните, что сущность, где действительно хранится значение свойства (значение double в случае Height), представляется как открытое, статическое, допускающее только чтение поле типа DependencyProperty. По соглашению имя этого свойства должно всегда формироваться из имени связанной оболочки CLR с добавлением суффикса Property:

```
public static readonly DependencyProperty HeightProperty;
```

Учитывая, что свойства зависимости объявляются как статические поля, они обычно создаются (и регистрируются) внутри статического конструктора класса. Объект DependencyProperty создается посредством вызова статического метода DependencyProperty.Register(). Данный метод имеет множество перегруженных версий, но в случае свойства Height он вызывается следующим образом:

```
HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata((double)0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
        new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
```

Первым аргументом, передаваемым методу DependencyProperty.Register(), является имя обычного свойства CLR класса (Height), а второй аргумент содержит информацию о типе данных, который его инкапсулирует (double). Третий аргумент указывает информацию о типе класса, которому принадлежит свойство (FrameworkElement). Хотя такие сведения могут показаться избыточными (в конце концов, поле HeightProperty уже определено внутри класса FrameworkElement), это довольно продуманный аспект WPF, поскольку он позволяет одному классу регистрировать свойства в другом классе (даже если его определение было запечатано).

Четвертый аргумент, передаваемый методу DependencyProperty.Register() в рассмотренном примере, представляет собой то, что действительно делает свойства зависимости уникальными. Здесь передается объект FrameworkPropertyMetadata, который описывает разнообразные детали относительно того, как инфраструктура

WPF должна обрабатывать данное свойство в плане уведомлений с помощью обратных вызовов (если свойству необходимо извещать других, когда его значение изменяется). Кроме того, объект FrameworkPropertyMetadata указывает различные параметры (представленные перечислением FrameworkPropertyMetadataOptions), которые управляют тем, на что свойство воздействует (работает ли оно с привязкой данных, может ли наследоваться и т.д.). В данном случае аргументы конструктора FrameworkPropertyMetadata можно описать так:

```
new FrameworkPropertyMetadata(
    // Стандартное значение свойства.
    (double)0.0,
    // Параметры метаданных.
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    // Делегат, который указывает на метод,
    // вызываемый при изменении свойства.
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

Поскольку последний аргумент конструктора FrameworkPropertyMetadata является делегатом, обратите внимание, что он указывает на статический метод OnTransformDirty() класса FrameworkElement. Код метода OnTransformDirty() здесь не приводится, но имейте в виду, что при создании специального свойства зависимости всегда можно указывать делегат PropertyChangedCallback, нацеленный на метод, который будет вызываться в случае изменения значения свойства.

Это подводит к финальному параметру метода DependencyProperty.Register() — второму делегату типа ValidateValueCallback, указывающему на метод класса FrameworkElement, который вызывается для проверки достоверности значения, присваиваемого свойству:

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

Метод IsWidthHeightValid() содержит логику, которую обычно ожидают найти в блоке установки значения свойства (как более подробно объясняется в следующем разделе):

```
private static bool IsWidthHeightValid(object value)
{
    double num = (double) value;
    return ((!DoubleUtil.IsNaN(num) && (num >= 0.0))
        && !double.IsPositiveInfinity(num));
}
```

После того, как объект DependencyProperty зарегистрирован, остается упаковать поле в обычное свойство CLR (Height в рассматриваемом случае). Тем не менее, обратите внимание, что блоки get и set не просто возвращают или устанавливают значение double переменной-члена уровня класса, а делают это косвенно с использованием методов GetValue() и SetValue() базового класса System.Windows.DependencyObject:

```
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```

Важные замечания относительно оболочек свойств CLR

Подводя итог, следует отметить, что свойства зависимости выглядят как обычные свойства, когда вы извлекаете или устанавливаете их значения в разметке XAML либо в коде, но “за кулисами” они реализованы с помощью гораздо более замысловатых программных приемов. Вспомните, что основным назначением этого процесса является построение специального элемента управления, имеющего специальные свойства, которые должны быть интегрированы со службами WPF, требующими взаимодействия через свойства зависимости (например, с анимацией, привязкой данных и стилями).

Несмотря на то что часть реализации свойства зависимости предусматривает определение оболочки CLR, вы никогда не должны помещать логику проверки достоверности в блок `set`. К тому же оболочка CLR свойства зависимости не должна делать ничего кроме вызовов `GetValue()` или `SetValue()`.

Исполняющая среда WPF сконструирована таким образом, что если написать разметку XAML, которая выглядит как установка свойства, например:

```
<Button x:Name="myButton" Height="100" .../>
```

то исполняющая среда вообще обойдет блок установки свойства `Height` и *напрямую* вызовет метод `SetValue()`! Причина такого необычного поведения связана с простым приемом оптимизации. Если бы исполняющая среда WPF обращалась к блоку установки свойства `Height`, то ей пришлось бы во время выполнения выяснить посредством рефлексии, где находится поле `DependencyProperty` (указанное в первом аргументе `SetValue()`), ссылаясь на него в памяти и т.д. То же самое остается справедливым и при написании разметки XAML, которая извлекает значение свойства `Height` — метод `GetValue()` будет вызываться напрямую. Но раз так, тогда зачем вообще строить оболочку CLR? Дело в том, что XAML в WPF не позволяет вызывать функции в разметке, поэтому следующий фрагмент приведет к ошибке:

```
<!-- Ошибка! Вызывать методы в XAML-разметке WPF нельзя! -->
<Button x:Name="myButton" this.SetValue("100") .../>
```

На самом деле установку или получение значения в разметке с применением оболочки CLR следует считать способом сообщения исполняющей среде WPF о необходимости вызова методов `GetValue()`/`SetValue()`, т.к. напрямую вызывать их в разметке невозможно. А что, если обратиться к оболочке CLR в коде, как показано ниже?

```
Button b = new Button();
b.Height = 10;
```

В таком случае, если блок `set` свойства `Height` содержит какой-то код помимо вызова `SetValue()`, то он *должен* выполниться, потому что оптимизация синтаксического анализатора XAML в WPF не задействуется.

Запомните основное правило: при регистрации свойства зависимости используйте делегат `ValidateValueCallback` для указания на метод, который выполняет проверку достоверности данных. Такой подход гарантирует корректное поведение независимо от того, что именно применяется для получения/установки свойства зависимости — разметка XAML или код.

Построение специального свойства зависимости

Если к настоящему моменту вы слегка запутались, то такая реакция совершенно нормальна. Создание свойств зависимости может требовать некоторого времени на

привыкание. Как бы то ни было, но это часть процесса построения многих специальных элементов управления WPF, так что давайте рассмотрим, каким образом создается свойство зависимости.

Начните с создания нового проекта приложения WPF по имени CustomDependencyProperty. Выберите в меню Project (Проект) пункт Add User Control (WPF) (Добавить пользовательский элемент управления (WPF)) и создайте элемент управления с именем ShowNumberControl.xaml.

На заметку! Более подробные сведения о классе UserControl в WPF ищите в главе 27, а пока просто следуйте указаниям по мере проработки примера.

Подобно окну типы UserControl в WPF имеют файл XAML и связанный файл кода. Модифицируйте разметку XAML пользовательского элемента управления, чтобы определить простой элемент Label внутри Grid:

```
<UserControl x:Class="CustomDependencyProperty.ShowNumberControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:CustomDependencyProperty"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Label x:Name="numberDisplay" Height="50" Width="200"
            Background="LightBlue"/>
    </Grid>
</UserControl>
```

В файле кода для данного элемента создайте обычное свойство .NET Core, которое упаковывает поле типа int и устанавливает новое значение для свойства Content элемента Label:

```
public partial class ShowNumberControl : UserControl
{
    public ShowNumberControl()
    {
        InitializeComponent();
    }

    // Обычное свойство .NET Core.
    private int _currNumber = 0;
    public int CurrentNumber
    {
        get => _currNumber;
        set
        {
            _currNumber = value;
            numberDisplay.Content = CurrentNumber.ToString();
        }
    }
}
```

Обновите определение XAML в MainWindow.xaml, объявив экземпляр специального элемента управления внутри диспетчера компоновки StackPanel. Поскольку специ-

альный элемент управления не входит в состав основных сборок WPF, понадобится определить специальное пространство имен XML, которое отображается на него. Вот требуемая разметка:

```
<Window x:Class="CustomDependencyProperty.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:myCtrls="clr-namespace: CustomDependencyProperty"
    xmlns:local="clr-namespace: CustomDependencyProperty"
    mc:Ignorable="d"
    Title="Simple Dependency Property App" Height="450" Width="450"
    WindowStartupLocation="CenterScreen">
<StackPanel>
    <myCtrls:ShowNumberControl
        HorizontalAlignment="Left" x:Name="myShowNumberCtrl"
        CurrentNumber="100"/>
</StackPanel>
</Window>
```

Похоже, что визуальный конструктор Visual Studio корректно отображает значение, установленное в свойстве CurrentNumber (рис. 25.23).

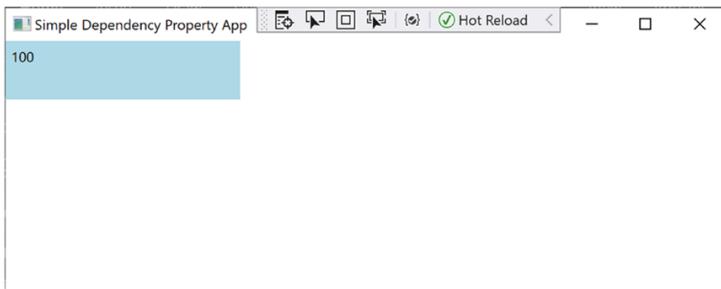


Рис. 25.23. Свойство выглядит работающим так, как ожидалось

Однако что, если к свойству CurrentNumber необходимо применить объект анимации, который обеспечит изменение значения свойства от 100 до 200 в течение 10 секунд? Если это желательно сделать в разметке, тогда область myCtrls:ShowNumberControl можно изменить следующим образом:

```
<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
    <myCtrls:ShowNumberControl.Triggers>
        <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard TargetProperty = "CurrentNumber">
                        <Int32Animation From = "100" To = "200" Duration = "0:0:10"/>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>
```

После запуска приложения объект анимации не сможет найти подходящую цель и генерируется исключение. Причина в том, что свойство CurrentNumber не было зарегистрировано как свойство зависимости! Чтобы устранить проблему, возвратитесь в файл кода для специального элемента управления и полностью закомментируйте текущую логику свойства (включая закрытое поддерживающее поле).

Теперь добавьте показанный ниже код, чтобы свойство CurrentNumber создавалось как свойство зависимости:

```
public int CurrentNumber
{
    get => (int)GetValue(CurrentNumberProperty);
    set => SetValue(CurrentNumberProperty, value);
}

public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
        typeof(int),
        typeof(ShowNumberControl),
        new UIPropertyMetadata(0));
```

Работа похожа на ту, что делалась в реализации свойства Height; тем не менее, предыдущий фрагмент кода регистрирует свойство непосредственно в теле, а не в статическом конструкторе (что хорошо). Также обратите внимание, что объект UIPropertyMetadata используется для определения стандартного целочисленного значения (0) вместо более сложного объекта FrameworkPropertyMetadata. В итоге получается простейшая версия CurrentNumber как свойства зависимости.

Добавление процедуры проверки достоверности данных

Хотя у вас есть свойство зависимости по имени CurrentNumber (и исключение больше не генерируется), анимация пока еще не наблюдается. Следующей корректировкой будет указание функции, вызываемой для выполнения проверки достоверности данных. В данном примере предполагается, что нужно обеспечить нахождение значения свойства CurrentNumber в диапазоне между 0 и 500.

Добавьте в метод DependencyProperty.Register() последний аргумент типа ValidateValueCallback, указывающий на метод по имени ValidateCurrentNumber.

Здесь ValidateValueCallback является делегатом, который может указывать только на методы, возвращающие тип bool и принимающие единственный аргумент типа object. Экземпляр object представляет присваиваемое новое значение. Реализация ValidateCurrentNumber должна возвращать true, если входное значение находится в ожидаемом диапазоне, и false в противном случае:

```
public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber",
        typeof(int),
        typeof(ShowNumberControl),
        new UIPropertyMetadata(100),
        new ValidateValueCallback(ValidateCurrentNumber));

// Простое бизнес-правило: значение должно находиться
// в диапазоне между 0 и 500.
public static bool ValidateCurrentNumber(object value) =>
    Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500;
```

Реагирование на изменение свойства

Итак, допустимое число уже есть, но анимация по-прежнему отсутствует. Последнее изменение, которое потребуется внести — передать во втором аргументе конструктора `UIPropertyMetadata` объект `PropertyChangedCallback`. Данный делегат может указывать на любой метод, принимающий `DependencyObject` в первом параметре и `DependencyPropertyChangedEventArgs` во втором. Модифицируйте код следующим образом:

```
// Обратите внимание на второй параметр конструктора UIPropertyMetadata.
public static readonly DependencyProperty CurrentNumberProperty =
    DependencyProperty.Register("CurrentNumber", typeof(int),
                               typeof(ShowNumberControl),
    new UIPropertyMetadata(100,
                           new PropertyChangedCallback(PropertyChanged)),
    new ValidateValueCallback(ValidateCurrentNumber));
```

Конечной целью внутри метода `CurrentNumberChamged()` будет изменение свойства `Content` объекта `Label` на новое значение, присвоенное свойству `CurrentNumber`. Однако возникает серьезная проблема: метод `CurrentNumberChanged()` является статическим, т.к. он должен работать со статическим объектом `DependencyProperty`. Как тогда получить доступ к объекту `Label` для текущего экземпляра `ShowNumberControl`? Нужная ссылка содержится в первом параметре `DependencyObject`. Новое значение можно найти с применением входных аргументов события. Ниже показан необходимый код, который будет изменять свойство `Content` объекта `Label`:

```
private static void CurrentNumberChanged(DependencyObject depObj,
                                         DependencyPropertyChangedEventArgs args)
{
    // Привести DependencyObject к ShowNumberControl.
    ShowNumberControl c = (ShowNumberControl)depObj;
    // Получить элемент управления Label в ShowNumberControl.
    Label theLabel = c.numberDisplay;
    // Установить для Label новое значение.
    theLabel.Content = args.NewValue.ToString();
}
```

Видите, насколько долгий путь пришлось пройти, чтобы всего лишь изменить содержащиеся метки! Преимущество заключается в том, что теперь свойство зависимости `CurrentNumber` может быть целью для стиля WPF, объекта анимации, операции привязки данных и т.д. Снова запустив приложение, вы легко заметите, что значение изменяется во время выполнения.

На этом обзор свойств зависимости WPF завершен. Хотя теперь вы должны гораздо лучше понимать, что они позволяют делать, и как создавать собственные свойства подобного рода, имейте в виду, что многие детали здесь не были раскрыты.

Если вам однажды понадобится создавать множество собственных элементов управления, поддерживающих специальные свойства, тогда загляните в подраздел “Properties” (“Свойства”) раздела “Systems” (“Системы”) документации по WPF (<https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/>). Там вы найдете намного больше примеров построения свойств зависимости, присоединяемых свойств, разнообразных способов конфигурирования метаданных и массу других подробных сведений.

Резюме

В главе рассматривались некоторые аспекты элементов управления WPF, начиная с обзора набора инструментов для элементов управления и роли диспетчеров компоновки (панелей). Первый пример был посвящен построению простого приложения текстового процессора. В нем демонстрировалось использование интегрированной в WPF функциональности проверки правописания, а также создание главного окна с системой меню, строкой состояния и панелью инструментов.

Более важно то, что вы научились строить команды WPF. Эти независимые от элементов управления события можно присоединять к элементу пользовательского интерфейса или входному жесту для автоматического наследования готовой функциональности (например, операций с буфером обмена).

Кроме того, вы узнали немало сведений о построении пользовательских интерфейсов в XAML и попутно ознакомились с интерфейсом Ink API, предлагаемым WPF. Вы также получили представление об операциях привязки данных WPF, включая использование класса DataGrid из WPF для отображения информации из специальной базы данных AutoLot.

Наконец, вы выяснили, что инфраструктура WPF добавляет уникальный аспект к традиционным программным примитивам .NET Core, в частности к свойствам и событиям. Как было показано, механизм свойств зависимости позволяет строить свойство, которое может интегрироваться с набором служб WPF (анимации, привязки данных, стили и т.д.). В качестве связанного замечания: механизм маршрутизируемых событий предоставляет событию способ распространяться вверх или вниз по дереву разметки.

ГЛАВА 26

Службы визуализации графики WPF

В настоящей главе рассматриваются возможности графической визуализации WPF. Вы увидите, что инфраструктура WPF предоставляет три отдельных способа визуализации графических данных: фигуры, рисунки и визуальные объекты. Разобравшись в преимуществах и недостатках каждого подхода, вы приступите к исследованию мира интерактивной двумерной графики с использованием классов из пространства имен `System.Windows.Shapes`. Затем будет показано, как с помощью рисунков и геометрических объектов визуализировать двумерные данные в легковесной манере. И, наконец, вы узнаете, каким образом добиться от визуального уровня максимальной функциональности и производительности.

Попутно затрагиваются многие связанные темы, такие как создание специальных кистей и перьев, применение графических трансформаций к визуализации и выполнение операций проверки попадания. В частности вы увидите, как можно упростить решение задач кодирования графики с помощью интегрированных инструментов Visual Studio и дополнительного средства под названием Inkscape.

На заметку! Графика является ключевым аспектом разработки WPF. Даже если вы не строите приложение с интенсивной графикой (вроде видеоигры или мультимедийного приложения), то рассматриваемые в главе темы критически важны при работе с такими службами, как шаблоны элементов управления, анимация и настройка привязки данных.

Понятие служб визуализации графики WPF

В WPF используется особая разновидность графической визуализации, которая известна под названием *графика режима сохранения* (*retained mode*). Выражаясь просто, это означает, что после применения разметки XAML или процедурного кода для генерирования графической визуализации инфраструктура WPF несет ответственность за сохранение визуальных элементов и обеспечение их корректной перерисовки и обновления оптимальным способом. Таким образом, визуализируемые графические данные присутствуют постоянно, даже когда конечный пользователь скрывает изображение, изменяя размер окна или сворачивая его, перекрывая одно окно другим и т.д.

По разительному контрасту предшествующие версии API-интерфейсов графической визуализации от Microsoft (включая GDI+ в Windows Forms) были графическими системами *прямого режима* (*immediate mode*). В такой модели ответственность за

корректное “запоминание” и обновление визуализируемых элементов на протяжении времени жизни приложения возлагалась на программиста. Например, в приложении Windows Forms визуализация фигуры вроде прямоугольника предусматривала обработку события Paint (или переопределение виртуального метода OnPaint()), получение объекта Graphics для рисования прямоугольника и, что важнее всего, добавление инфраструктуры, обеспечивающей сохранение изображения в ситуации, когда пользователь изменил размеры окна (например, за счет создания переменных-членов для представления позиции прямоугольника и вызова метода Invalidate() во многих местах кода).

Переход от графики прямого режима к графике режима сохранения — действительно удачное решение, т.к. программистам приходится писать и сопровождать гораздо меньший объем рутинного кода для поддержки графики. Однако речь не идет о том, что API-интерфейс графики WPF полностью отличается от более ранних инструментальных наборов визуализации. Например, как и GDI+, инфраструктура WPF поддерживает разнообразные типы объектов кистей и перьев, приемы проверки попадания, области отсечения, графические трансформации и т.д. Поэтому если у вас есть опыт работы с GDI+ (или GDI на языке C/C++), то вы уже имеете неплохое представление о способе выполнения базовой визуализации в WPF.

Варианты графической визуализации WPF

Как и с другими аспектами разработки приложений WPF, существует выбор из нескольких способов выполнения графической визуализации после принятия решения делать это посредством разметки XAML или процедурного кода C# (либо их комбинации). В частности, инфраструктура WPF предлагает следующие три индивидуальных подхода к визуализации графических данных.

- **Фигуры.** Инфраструктура WPF предоставляет пространство имен System.Windows.Shapes, в котором определено небольшое количество классов для визуализации двумерных геометрических объектов (прямоугольников, эллипсов, многоугольников и т.п.). Хотя такие типы просты в использовании и очень мощные, в случае непродуманного применения они могут привести к значительным накладным расходам памяти.
- **Рисунки и геометрические объекты.** Второй способ визуализации графических данных в WPF предполагает работу с классами, производными от абстрактного класса System.Windows.Media.Drawing. Используя классы, подобные GeometryDrawing или ImageDrawing (в дополнение к различным геометрическим объектам), можно визуализировать графические данные в более легковесной (но менее функциональной) манере.
- **Визуальные объекты.** Самый быстрый и легковесный способ визуализации графических данных в WPF предусматривает работу с визуальным уровнем, который доступен только через код C#. С применением классов, производных от System.Windows.Media.Visual, можно взаимодействовать непосредственно с графической подсистемой WPF.

Причина предоставления разных способов решения той же самой задачи (т.е. визуализации графических данных) связана с расходом памяти и в конечном итоге с производительностью приложения. Поскольку WPF является системой, интенсивно использующей графику, нет ничего необычного в том, что приложению требуется ви-

зуализировать сотни или даже тысячи различных изображений на поверхности окна, и выбор реализации (фигуры, рисунки или визуальные объекты) может оказать огромное влияние.

Важно понимать, что при построении приложения WPF высока вероятность использования всех трех подходов. В качестве эмпирического правила запомните: если нужен умеренный объем интерактивных графических данных, которыми может манипулировать пользователь (принимающих ввод от мыши, отображающих всплывающие подсказки и т.д.), то следует применять члены из пространства имен `System.Windows.Shapes`.

Напротив, рисунки и геометрические объекты лучше подходят, когда необходимо моделировать сложные и по большей части не интерактивные векторные графические данные с использованием разметки XAML или кода C#. Хотя рисунки и геометрические объекты способны реагировать на события мыши, а также поддерживают проверку попадания и операции перетаскивания, для выполнения таких действий обычно приходится писать больше кода.

Наконец, если требуется самый быстрый способ визуализации значительных объемов графических данных, то должен быть выбран визуальный уровень. Например, предположим, что инфраструктура WPF применяется для построения научного приложения, которое должно отображать тысячи точек на графике данных. За счет использования визуального уровня точки на графике можно визуализировать оптимальным образом. Как будет показано далее в главе, визуальный уровень доступен только из кода C#, но не из разметки XAML.

Независимо от выбранного подхода (фигуры, рисунки и геометрические объекты или визуальные объекты) всегда будут применяться распространенные графические примитивы, такие как кисти (для заполнения ограниченных областей), перья (для рисования контуров) и объекты трансформаций (которые видоизменяют данные). Исследование начинается с классов из пространства имен `System.Windows.Shapes`.

На заметку! Инфраструктура WPF поставляется также с полнофункциональным API-интерфейсом, который можно использовать для визуализации и манипулирования трехмерной графикой, но в книге он не рассматривается.

Визуализация графических данных с использованием фигур

Члены пространства имен `System.Windows.Shapes` предлагают наиболее прямолинейный, интерактивный и самый затратный в плане расхода памяти способ визуализации двумерного изображения. Это небольшое пространство имен (расположенное в сборке `PresentationFramework.dll`) состоит всего из шести запечатанных классов, которые расширяют абстрактный базовый класс `Shape`: `Ellipse`, `Rectangle`, `Line`, `Polygon`, `Polyline` и `Path`.

Абстрактный класс `Shape` унаследован от класса `FrameworkElement`, который сам унаследован от `UIElement`. В указанных классах определены члены для работы с изменением размеров, всплывающими подсказками, курсорами мыши и т.п. Благодаря такой цепочке наследования при визуализации графических данных с применением классов, производных от `Shape`, объекты получаются почти такими же функциональными (с точки зрения взаимодействия с пользователем), как элементы управления WPF.

Скажем, для выяснения, щелкнул ли пользователь на визуализированном изображении, достаточно обработать событие `MouseDown`. Например, если написать следующую разметку XAML для объекта `Rectangle` внутри элемента управления `Grid` начального окна `Window`:

```
<Rectangle x:Name="myRect" Height="30" Width="30" Fill="Green"
    MouseDown="myRect_MouseDown"/>
```

то можно реализовать обработчик события `MouseDown`, который изменяет цвет фона прямоугольника в результате щелчка на нем:

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
    // Изменить цвет прямоугольника в результате щелчка на нем.
    myRect.Fill = Brushes.Pink;
}
```

В отличие от других инструментальных наборов, предназначенных для работы с графикой, вам не придется писать громоздкий код инфраструктуры, в котором вручную сопоставляются координаты мыши с геометрическим объектом, выясняется попадание курсора внутрь границ, выполняется визуализация в неотображаемый буфер и т.д. Члены пространства имен `System.Windows.Shapes` просто реагируют на зарегистрированные вами события подобно типичному элементу управления WPF (`Button` и т.д.).

Недостаток всей готовой функциональности связан с тем, что фигуры потребляют довольно много памяти. Если строится научное приложение, которое рисует тысячи точек на экране, то использование фигур будет неудачным выбором (по существу таким же расточительным в плане памяти, как визуализация тысяч объектов `Button`). Тем не менее, когда нужно генерировать интерактивное двумерное векторное изображение, фигуры оказываются прекрасным вариантом.

Помимо функциональности, унаследованной от родительских классов `UIElement` и `FrameworkElement`, в классе `Shape` определено множество собственных членов, наиболее полезные из которых кратко описаны в табл. 26.1.

На заметку! Если вы забудете установить свойства `Fill` и `Stroke`, то WPF предоставит "невидимые" кисти, вследствие чего фигура не будет видна на экране!

Добавление прямоугольников, эллипсов и линий на поверхность `Canvas`

Вы построите приложение WPF, которое способно визуализировать фигуры, с применением XAML и C#, и попутно исследуете процесс проверки попадания. Создайте новый проект приложения WPF по имени `RenderingWithShapes` и измените заголовок главного окна в `MainWindow.xaml` на `Fun with Shapes!`. Модифицируйте первоначальную разметку XAML для элемента `Window`, заменив `Grid` панелью `DockPanel`, которая содержит (пока пустые) элементы `ToolBar` и `Canvas`. Обратите внимание, что каждому содержащемуся элементу посредством свойства `Name` назначается подходящее имя.

```
<DockPanel LastChildFill="True">
    <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
        </ToolBar>
    <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

Таблица 26.1. Ключевые свойства базового класса Shape

Свойства	Описание
DefiningGeometry	Возвращает объект Geometry, который представляет общие размеры текущей фигуры. Этот объект содержит только точки, применяемые для визуализации данных, и не имеет никаких следов функциональности из класса UIElement или FrameworkElement
Fill	Позволяет указать объект кисти для заполнения внутренней области фигуры
GeometryTransform	Позволяет применять трансформацию к фигуре до ее визуализации на экране. Унаследованное (от UIElement) свойство RenderTransform применяет трансформацию <i>после</i> визуализации фигуры на экране
Stretch	Описывает, каким образом расположить фигуру в выделенном ей пространстве, например, по ее позиции внутри диспетчера компоновки. Это управляется с использованием соответствующего перечисления System.Windows.Media.Stretch
Stroke	Определяет объект кисти или в ряде случаев объект пера (который на самом деле является замаскированным объектом кисти), применяемый для рисования границы фигуры
StrokeDashArray, StrokeEndLineCap, StrokeStartLineCap, StrokeThickness	Эти (и другие) свойства, связанные со штрихами, управляют тем, как сконфигурированы линии при рисовании границ фигуры. В большинстве случаев данные свойства будут конфигурировать кисть, используемую для рисования границы или линии

Заполните элемент ToolBar набором объектов RadioButton, каждый из которых содержит объект специфического класса, производного от Shape. Легко заметить, что каждому элементу RadioButton назначается то же самое групповое имя GroupName (чтобы обеспечить взаимное исключение) и также подходящее индивидуальное имя.

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
    <RadioButton Name="circleOption" GroupName="shapeSelection"
        Click="CircleOption_Click">
        <Ellipse Fill="Green" Height="35" Width="35" />
    </RadioButton>
    <RadioButton Name="rectOption" GroupName="shapeSelection"
        Click="RectOption_Click">
        <Rectangle Fill="Red" Height="35" Width="35" RadiusY="10" RadiusX="10" />
    </RadioButton>
    <RadioButton Name="lineOption" GroupName="shapeSelection"
        Click="LineOption_Click">
        <Line Height="35" Width="35" StrokeThickness="10" Stroke="Blue"
            X1="10" Y1="10" Y2="25" X2="25"
            StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
    </RadioButton>
</ToolBar>
```

Как видите, объявление объектов Rectangle, Ellipse и Line в разметке XAML довольно прямолинейно и требует лишь минимальных комментариев. Вспомните, что

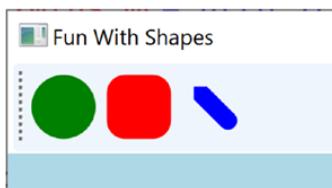


Рис. 26.1. Применение объектов Shape в качестве содержимого для набора элементов RadioButton

при описании линии имеют мало смысла). Здесь устанавливается несколько дополнительных свойств, которые управляют тем, как визуализируются начальная и конечная точки объекта Line, а также настраивают параметры штриха. На рис. 26.1 показана визуализированная панель инструментов визуальном конструкторе WPF среди Visual Studio.

С помощью окна Properties (Свойства) среди Visual Studio создайте обработчик события MouseLeftButtonDown для Canvas и обработчик события Click для каждого элемента RadioButton. Цель заключается в том, чтобы в коде C# визуализировать выбранную фигуру (круг, квадрат или линию), когда пользователь щелкает внутри Canvas. Первым делом определите следующее вложенное перечисление (и соответствующую переменную-член) внутри класса, производного от Window:

```
public partial class MainWindow : Window
{
    private enum SelectedShape
    { Circle, Rectangle, Line }
    private SelectedShape _currentShape;
    ...
}
```

В каждом обработчике Click установите переменную-член currentShape в корректное значение SelectedShape:

```
private void CircleOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Circle;
}

private void RectOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Rectangle;
}

private void LineOption_Click(object sender, RoutedEventArgs e)
{
    _currentShape = SelectedShape.Line;
}
```

Посредством обработчика события MouseLeftButtonDown элемента Canvas будет визуализироваться подходящая фигура (предопределенного размера) в начальной точке, которая соответствует позиции (x, y) курсора мыши. Ниже приведена полная реализация с последующим анализом:

свойство Fill позволяет указать кисть для рисования внутренностей фигуры. Когда нужна кисть сплошного цвета, можно просто задать жестко закодированную строку известных значений, а соответствующий преобразователь типа генерирует корректный объект. Интересная характеристика типа Rectangle связана с тем, что в нем определены свойства RadiusX и RadiusY, позволяющие визуализировать скругленные углы.

Объект Line представлен своими начальной и конечной точками с использованием свойств X1, X2, Y1 и Y2 (учитывая, что высота и ширина

```

private void CanvasDrawingArea_MouseLeftButtonDown(object sender,
    MouseEventArgs e)
{
    Shape shapeToRender = null;
    // Сконфигурировать корректную фигуру для рисования.
    switch (_currentShape)
    {
        case SelectedShape.Circle:
            shapeToRender = new Ellipse() { Fill = Brushes.Green,
                Height = 35, Width = 35 };
            break;
        case SelectedShape.Rectangle:
            shapeToRender = new Rectangle()
            { Fill = Brushes.Red, Height = 35, Width = 35,
                RadiusX = 10, RadiusY = 10 };
            break;
        case SelectedShape.Line:
            shapeToRender = new Line()
            {
                Stroke = Brushes.Blue,
                StrokeThickness = 10,
                X1 = 0, X2 = 50, Y1 = 0, Y2 = 50,
                StrokeStartLineCap= PenLineCap.Triangle,
                StrokeEndLineCap = PenLineCap.Round
            };
            break;
        default:
            return;
    }
    // Установить левый верхний угол для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}

```

На заметку! Возможно, вы заметили, что объекты `Ellipse`, `Rectangle` и `Line`, создаваемые в методе `canvasDrawingArea_MouseLeftButtonDown()`, имеют те же настройки свойств, что и соответствующие определения XAML. Вполне ожидаемо, код можно упростить, но это требует понимания объектных ресурсов WPF, которые будут рассматриваться в главе 27.

В коде проверяется переменная-член `_currentShape` с целью создания корректного объекта, производного от `Shape`. Затем устанавливаются координаты левого верхнего угла внутри `Canvas` с использованием входного объекта `MouseEventArgs`. Наконец, в коллекцию объектов `UIElement`, поддерживаемую `Canvas`, добавляется новый производный от `Shape` объект. Если запустить программу прямо сейчас, то она должна позволить щелкать левой кнопкой мыши где угодно на холсте и визуализировать в позиции щелчка выбранную фигуру.

Удаление прямоугольников, эллипсов и линий с поверхности Canvas

Имея в распоряжении элемент Canvas с коллекцией объектов, может возникнуть вопрос: как динамически удалить элемент, скажем, в ответ на щелчок пользователя правой кнопкой мыши на фигуре? Это делается с помощью класса VisualTreeHelper из пространства имен System.Windows.Media. Роль “визуальных деревьев” и “логических деревьев” более подробно объясняется в главе 27, а пока организуйте обработку события MouseRightButtonDown объекта Canvas и реализуйте соответствующий обработчик:

```
private void CanvasDrawingArea_MouseRightButtonDown(object sender,
    MouseButtonEventArgs e)
{
    // Сначала получить координаты x, y позиции,
    // где пользователь выполнил щелчок.
    Point pt = e.GetPosition((Canvas)sender);

    // Использовать метод HitTest() класса VisualTreeHelper, чтобы
    // выяснить, щелкнул ли пользователь на элементе внутри Canvas.
    HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);

    // Если переменная result не равна null, то щелчок произведен на фигуре.
    if (result != null)
    {
        // Получить фигуру, на которой совершен щелчок, и удалить ее из Canvas.
        canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
    }
}
```

Метод начинается с получения точных координат (x, y) позиции, где пользователь щелкнул внутри Canvas, и проверки попадания посредством статического метода VisualTreeHelper.HitTest(). Возвращаемое значение — объект HitTestResult — будет установлено в null, если пользователь выполнил щелчок не на UIElement внутри Canvas. Если значение HitTestResult не равно null, тогда с помощью свойства VisualHit можно получить объект UIElement, на котором был совершен щелчок, и привести его к типу, производному от Shape (вспомните, что Canvas может содержать любой UIElement, а не только фигуры). Детали, связанные с “визуальным деревом”, будут изложены в главе 27.

На заметку! По умолчанию метод VisualTreeHelper.HitTest() возвращает объект UIElement самого верхнего уровня, на котором совершен щелчок, и не предоставляет информацию о других объектах, расположенных под ним (т.е. перекрытых в Z-порядке).

В результате внесенных модификаций должна появиться возможность добавления фигуры на Canvas щелчком левой кнопкой мыши и ее удаления щелчком правой кнопкой мыши.

До настоящего момента вы применяли объекты типов, производных от Shape, для визуализации содержимого элементов RadioButton с использованием разметки XAML и заполняли Canvas в коде C#. Во время исследования роли кистей и графических трансформаций в данный пример будет добавлена дополнительная функциональность. К слову, в другом примере главы будут иллюстрироваться приемы пере-

таскивания на объектах `UIElement`. А пока давайте рассмотрим оставшиеся члены пространства имен `System.Windows.Shapes`.

Работа с элементами `Polyline` и `Polygon`

В текущем примере используются только три класса, производных от `Shape`. Остальные дочерние классы (`Polyline`, `Polygon` и `Path`) чрезвычайно трудно корректно визуализировать без инструментальной поддержки (такой как инструмент Blend для Visual Studio или другие инструменты, которые могут создавать векторную графику) — просто потому, что они требуют определения большого количества точек для своего выходного представления. Ниже представлен краткий обзор остальных типов `Shapes`.

Тип `Polyline` позволяет определить коллекцию координат (*x*, *y*) (через свойство `Points`) для рисования последовательности линейных сегментов, не требующих замыкания. Тип `Polygon` похож, но запрограммирован так, что всегда замыкает контур, соединяя начальную точку с конечной, и заполняет внутреннюю область с помощью указанной кисти. Предположим, что в редакторе Kaxaml создан следующий элемент `StackPanel`:

```
<!-- Элемент Polyline не замыкает автоматически конечные точки -->
<Polyline Stroke ="Red" StrokeThickness ="20" StrokeLineJoin ="Round"
    Points ="10,10 40,40 10,90 300,50"/>
<!-- Элемент Polygon всегда замыкает конечные точки -->
<Polygon Fill ="AliceBlue" StrokeThickness ="5" Stroke ="Green"
    Points ="40,10 70,80 10,50" />
```

На рис. 26.2 показан визуализированный вывод в Kaxaml.

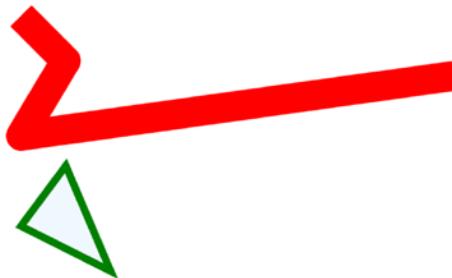


Рис. 26.2. Элементы `Polyline` и `Polygon`

Работа с элементом `Path`

Применяя только типы `Rectangle`, `Ellipse`, `Polygon`, `Polyline` и `Line`, нарисовать детализированное двумерное векторное изображение было бы исключительно трудно, т.к. упомянутые примитивы не позволяют легко фиксировать графические данные, подобные кривым, объединениям перекрывающихся данных и т.д. Последний производный от `Shape` класс, `Path`, предоставляет возможность определения сложных двумерных графических данных в виде коллекции независимых геометрических объектов. После того, как коллекция таких геометрических объектов определена, ее можно присвоить свойству `Data` класса `Path`, где она будет использоваться для визуализации сложного двумерного изображения.

Свойство `Data` получает объект класса, производного от `System.Windows.Media.Geometry`, который содержит ключевые члены, кратко описанные в табл. 26.2.

Таблица 26.2. Избранные члены класса `System.Windows.Media.Geometry`

Член	Описание
<code>Bounds</code>	Устанавливает текущий ограничивающий прямоугольник, который содержит геометрический объект
<code>FillContains()</code>	Выясняет, находится ли заданный объект <code>Point</code> (или другой объект <code>Geometry</code>) внутри границ определенного класса, производного от <code>Geometry</code> . Это полезно при вычислениях для проверки попадания
<code>GetArea()</code>	Возвращает общую область, занятую объектом производного от <code>Geometry</code> типа
<code>GetRenderBounds()</code>	Возвращает объект <code>Rect</code> , содержащий наименьший из возможных прямоугольник, который может быть применен для визуализации объекта класса, производного от <code>Geometry</code>
<code>Transform</code>	Назначает геометрическому объекту экземпляр класса <code>Transform</code> для изменения визуализации

Классы, которые расширяют класс `Geometry` (табл. 26.3), выглядят очень похожими на свои аналоги, производные от `Shape`. Например, класс `EllipseGeometry` имеет члены, подобные членам класса `Ellipse`. Крупное отличие связано с тем, что производные от `Geometry` классы не знают, каким образом визуализировать себя напрямую, поскольку они не являются `UIElement`. Взамен классы, производные от `Geometry`, представляют всего лишь коллекцию данных о точках, которая указывает объекту `Path`, как их визуализировать.

Таблица 26.3. Классы, производные от `Geometry`

Класс	Описание
<code>LineGeometry</code>	Представляет прямую линию
<code>RectangleGeometry</code>	Представляет прямоугольник
<code>EllipseGeometry</code>	Представляет эллипс
<code>GeometryGroup</code>	Позволяет группировать вместе несколько объектов <code>Geometry</code>
<code>CombinedGeometry</code>	Позволяет объединять два разных объекта <code>Geometry</code> в единую фигуру
<code>PathGeometry</code>	Представляет фигуру, образованную из линий и кривых

На заметку! Класс `Path` не является единственным классом в инфраструктуре WPF, который способен работать с коллекцией геометрических объектов. Например, классы `DoubleAnimationUsingPath`, `DrawingGroup`, `GeometryDrawing` и даже `UIElement` могут использовать геометрические объекты для визуализации с применением свойств `PathGeometry`, `ClipGeometry`, `Geometry` и `Clip` соответственно.

В показанной далее разметке для элемента `Path` используется несколько типов, производных от `Geometry`. Обратите внимание, что свойство `Data` объекта `Path` у-

танавливается в объект GeometryGroup, который содержит объекты других производных от Geometry классов, таких как EllipseGeometry, RectangleGeometry и LineGeometry. Результат представлен на рис. 26.3.

```
<!-- Элемент Path содержит набор объектов
    Geometry, установленный в свойстве Data -->
<Path Fill = "Orange" Stroke = "Blue"
    StrokeThickness = "3">
    <Path.Data>
        <GeometryGroup>
            <EllipseGeometry Center = "75,70"
                RadiusX = "30" RadiusY = "30" />
            <RectangleGeometry Rect = "25,55 100 30" />
            <LineGeometry StartPoint="0,0" EndPoint="70,30" />
            <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

Изображение на рис. 26.3 может быть визуализировано с применением показанных ранее классов Line, Ellipse и Rectangle. Однако это потребовало бы помещения различных объектов UIElement в память. Когда для моделирования точек рисуемого изображения используются геометрические объекты, а затем коллекция геометрических объектов помещается в контейнер, который способен визуализировать данные (Path в рассматриваемом случае), то тем самым сокращается расход памяти.

Теперь вспомните, что класс Path имеет ту же цепочку наследования, что и любой член пространства имен System.Windows.Shapes, а потому обладает возможностью отправлять такие же уведомления о событиях, как другие объекты UIElement. Следовательно, если определить тот же самый элемент <Path> в проекте Visual Studio, тогда выяснить, что пользователь щелкнул в любом месте линии, можно будет за счет обработки события мыши (не забывайте, что редактор XAML не разрешает обрабатывать события для написанной разметки).

“Мини-язык” моделирования путей

Из всех классов, перечисленных в табл. 26.3, класс PathGeometry наиболее сложен для конфигурирования в терминах XAML и кода. Причина объясняется тем фактом, что каждый *сегмент* PathGeometry состоит из объектов, содержащих разнообразные сегменты и фигуры (скажем, ArcSegment, BezierSegment, LineSegment, PolyBezierSegment, PolyLineSegment, PolyQuadraticBezierSegment и т.д.). Вот пример объекта Path, свойство Data которого было установлено в элемент PathGeometry, состоящий из различных фигур и сегментов:

```
<Path Stroke="Black" StrokeThickness="1">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigure StartPoint="10,50">
                    <PathFigure.Segments>
                        <BezierSegment
                            Point1="100,0"
                            Point2="200,200"
```

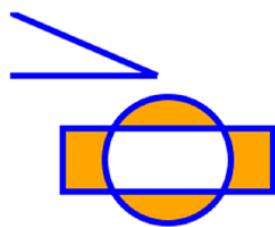


Рис. 26.3. Элемент Path, содержащий разнообразные объекты Geometry

```

    Point3="300,100"/>
<LineSegment Point="400,100" />
<ArcSegment
    Size="50,50" RotationAngle="45"
    IsLargeArc="True" SweepDirection="Clockwise"
    Point="200,100"/>
</PathFigure.Segments>
</PathFigure>
</PathGeometry.Figures>
</PathGeometry>
</Path.Data>
</Path>
```

По правде говоря, лишь немногим программистам придется когда-либо вручную строить сложные двумерные изображения, напрямую описывая объекты производных от `Geometry` или `PathSegment` классов. Позже в главе вы узнаете, как преобразовывать векторную графику в операторы “мини-языка” моделирования путей, которые можно применять в разметке XAML.

Даже с учетом содействия со стороны упомянутых ранее инструментов объем разметки XAML, требуемой для определения сложных объектов `Path`, может быть устраивающе большим, т.к. данные состоят из полных описаний различных объектов классов, производных от `Geometry` или `PathSegment`. Для того чтобы создавать более лаконичную разметку, в классе `Path` поддерживается специализированный “мини-язык”.

Например, вместо установки свойства `Data` объекта `Path` в коллекцию объектов классов, производных от `Geometry` и `PathSegment`, его можно установить в одиночный строковый литерал, содержащий набор известных символов и различных значений, которые определяют фигуру, подлежащую визуализации. Ниже приведен простой пример, а его результирующий вывод показан на рис. 26.4:

```

<Path Stroke="Black" StrokeThickness="3"
      Data="M 10,75 C 70,15 250,270 300,175 H 240" />
```

Команда `M` (от *move* — переместить) принимает координаты (*x*, *y*) позиции, которая представляет начальную точку рисования. Команда `C` (от *curve* — кривая) принимает последовательность точек для визуализации кривой (точнее кубической кривой Безье), а команда `H` (от *horizontal* — горизонталь) рисует горизонтальную линию.

И снова следует отметить, что вам очень редко придется вручную строить или анализировать строковый литерал, содержащий инструкции мини-языка моделирования путей. Тем не менее, цель в том, чтобы разметка XAML, генерируемая специализированными инструментами, не казалась вам совершенно непонятной.

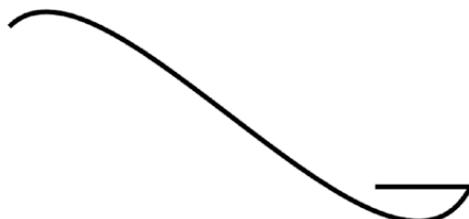


Рис. 26.4. “Мини-язык” моделирования путей позволяет компактно описывать объектную модель `Geometry/PathSegment`

Кисти и перья WPF

Каждый способ графической визуализации (фигуры, рисование и геометрические объекты, а также визуальные объекты) интенсивно использует *кисти*, которые позволяют управлять заполнением внутренней области двумерной фигуры. Инфраструктура WPF предлагает шесть разных типов кистей, и все они расширяют класс `System.Windows.Media.Brush`. Несмотря на то что `Brush` является абстрактным классом, его потомки, описанные в табл. 26.4, могут применяться для заполнения области содержимым почти любого мыслимого вида.

Таблица 26.4. Классы, производные от `Brush`

Класс	Описание
<code>DrawingBrush</code>	Заполняет область с помощью объекта производного от <code>Drawing</code> класса (<code>GeometryDrawing</code> , <code>ImageDrawing</code> или <code>VideoDrawing</code>)
<code>ImageBrush</code>	Заполняет область изображением (представленным посредством объекта <code>ImageSource</code>)
<code>LinearGradientBrush</code>	Заполняет область линейным градиентом
<code>RadialGradientBrush</code>	Заполняет область радиальным градиентом
<code>SolidColorBrush</code>	Заполняет область сплошным цветом, указанным в свойстве <code>Color</code>
<code>VisualBrush</code>	Заполняет область с помощью объекта производного от <code>Visual</code> класса (<code>DrawingVisual</code> , <code>Viewport3DVisual</code> и <code>ContentVisual</code>)

Классы `DrawingBrush` и `VisualBrush` позволяют строить кисть на основе существующего класса, производного от `Drawing` или `Visual`. Такие классы кистей используются при работе с двумя другими способами визуализации графики WPF (рисунками или визуальными объектами) и будут объясняться далее в главе.

Класс `ImageBrush` позволяет строить кисть, отображающую данные изображения из внешнего файла или встроенного ресурса приложения, который указан в его свойстве `ImageSource`. Оставшиеся типы кистей (`LinearGradientBrush` и `RadialGradientBrush`) довольно просты в применении, хотя требуемая разметка XAML может оказаться многословной. К счастью, в среде Visual Studio поддерживаются интегрированные редакторы кистей, которые облегчают задачу генерации стилизованных кистей.

Конфигурирование кистей с использованием Visual Studio

Давайте обновим приложение WPF для рисования `RenderingShapes`, чтобы использовать в нем более интересные кисти. В трех фигурах, которые были задействованы до сих пор при визуализации данных в панели инструментов, применяются обычные сплошные цвета, так что их значения можно зафиксировать с помощью простых строковых литералов. Чтобы сделать задачу чуть более интересной, теперь вы будете использовать интегрированный редактор кистей. Удостоверьтесь в том, что в IDE-среде открыт редактор XAML для начального окна и выберите элемент `Ellipse`. В окне `Properties` отыщите категорию `Brush` (Кисть) и щелкните на свойстве `Fill` (рис. 26.5).

В верхней части редактора кистей находится набор свойств, которые являются “совместимыми с кистью” для выбранного элемента (т.е. Fill, Stroke и OpacityMask). Под ними расположен набор вкладок, которые позволяют конфигурировать разные типы кистей, включая текущую кисть со сплошным цветом. Для управления цветом текущей кисти можно применять инструмент выбора цвета, а также ползунки ARGB (alpha, red, green, blue — прозрачность, красный, зеленый, синий). С помощью этих ползунков и связанный с ними области выбора цвета можно создавать сплошной цвет любого вида. Используйте указанные инструменты для изменения цвета в свойстве Fill элемента Ellipse и просмотрите результирующую разметку XAML. Как видите, цвет сохраняется в виде шестнадцатеричного значения:

```
<Ellipse Fill="#FF47CE47" Height="35"
        Width="35" />
```

Что более интересно, тот же самый редактор позволяет конфигурировать и градиентные кисти, которые применяются для определения последовательностей цветов и точек перехода цветов. Вспомните, что редактор кистей предлагает набор вкладок, первая из которых позволяет установить пустую кисть для отсутствующего визуализированного вывода. Остальные четыре дают возможность установить кисть сплошного цвета (как только что было показано), градиентную кисть, мозаичную кисть и кисть с изображением.

Щелкните на вкладке градиентной кисти; редактор отобразит несколько новых настроек (рис. 26.6).

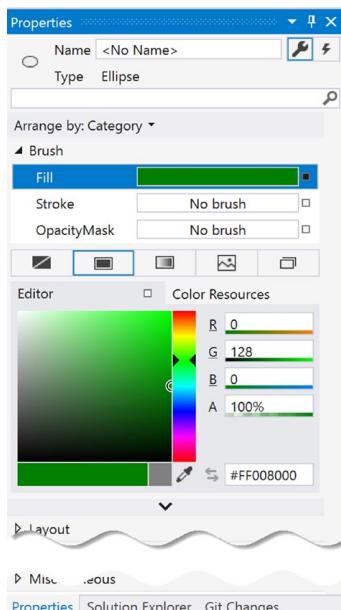


Рис. 26.5. Любое свойство, которое требует кисти, может быть сконфигурировано с помощью интегрированного редактора кистей

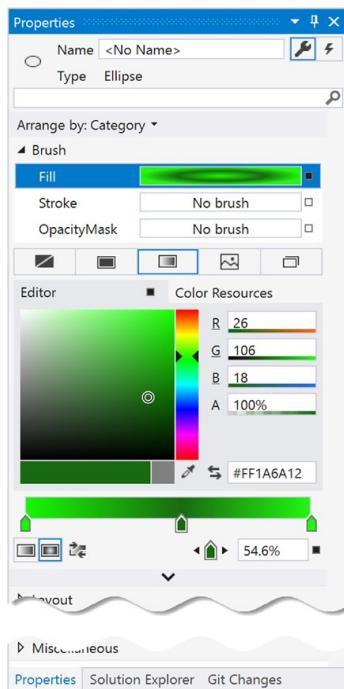


Рис. 26.6. Редактор кистей Visual Studio позволяет строить базовые градиентные кисти

Три кнопки в левом нижнем углу позволяют выбрать линейный градиент, радиальный градиент или обратить градиентные переходы. Полоса внизу покажет текущий цвет каждого градиентного перехода, который будет представлен специальным ползунком. Перетаскивая ползунок по полосе градиента, можно управлять смещением градиента. Кроме того, щелкая на конкретном ползунке, можно изменять цвет определенного градиентного перехода с помощью селектора цвета. Наконец, щелчок прямо на полосе градиента позволяет добавлять градиентные переходы.

Потратьте некоторое время на освоение этого редактора, чтобы построить радиальную градиентную кисть, содержащую три градиентных перехода, и установить их цвета. На рис. 26.6 показан пример кисти, использующей три оттенка зеленого цвета.

В результате IDE-среда обновит разметку XAML, добавив набор специальных кистей и присвоив им совместимым с кистями свойствам (свойство Fill элемента Ellipse в рассматриваемом примере) с применением синтаксиса “свойство-элемент”:

```
<Ellipse Height="35" Width="35">
<Ellipse.Fill>
  <RadialGradientBrush>
    <GradientStop Color="#FF17F800"/>
    <GradientStop Color="#FF24F610" Offset="1"/>
    <GradientStop Color="#FF1A6A12" Offset="0.546"/>
  </RadialGradientBrush>
</Ellipse.Fill>
</Ellipse>
```

Конфигурирование кистей в коде

Теперь, когда вы построили специальную кисть для определения XAML элемента Ellipse, соответствующий код C# устарел в том плане, что он по-прежнему будет визуализировать круг со сплошным зеленым цветом. Для восстановления синхронизации модифицируйте нужный оператор case, чтобы использовать только что созданную кисть. Ниже показано необходимое обновление, которое выглядит более сложным, чем можно было ожидать, т.к. шестнадцатеричное значение преобразуется в подходящий объект Color посредством класса System.Windows.

Media.ColorConverter (результат изменения представлен на рис. 26.7):

```
case SelectedShape.Circle:
  shapeToRender = new Ellipse() { Height = 35, Width = 35 };
  // Создать кисть RadialGradientBrush в коде.
  RadialGradientBrush brush = new RadialGradientBrush();
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF77F177"), 0));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF11E611"), 1));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF5A8E5A"), 0.545));
  shapeToRender.Fill = brush;
  break;
```

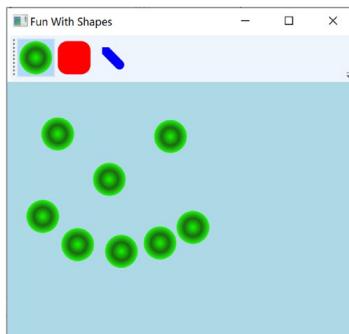


Рис. 26.7. Рисование более интересных кругов

Кстати, объекты `GradientStop` можно строить, указывая простой цвет в качестве первого параметра конструктора с применением перечисления `Colors`, которое дает сконфигурированный объект `Color`:

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

Если требуется более тонкий контроль, то можно передавать объект `Color`, сконфигурированный в коде, например:

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

Разумеется, использование перечисления `Colors` и класса `Color` не ограничивается градиентными кистями. Их можно применять всякий раз, когда необходимо представить значение цвета в коде.

Конфигурирование перьев

В сравнении с кистями *перо* представляет собой объект для рисования границ геометрических объектов или в случае класса `Line` либо `PolyLine` — самого линейного геометрического объекта. В частности, класс `Pen` позволяет рисовать линию указанной толщины, представленной значением типа `double`. Вдобавок объект `Pen` может быть сконфигурирован с помощью того же самого вида свойств, что и в классе `Shape`, таких как начальный и конечный концы пера, шаблоны точек-тире и т.д. Например, для определения атрибутов пера к определению фигуры можно добавить следующую разметку:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle"
      StartLineCap="Round" />
```

Во многих случаях создавать объект `Pen` непосредственно не придется, потому что это делается косвенно, когда присваиваются значения свойствам вроде `StrokeThickness` производного от `Shape` типа (а также других типов `UIElement`). Однако строить специальный объект `Pen` удобно при работе с типами, производными от `Drawing` (которые рассматриваются позже в главе). Среда Visual Studio не располагает редактором перьев как таковым, но позволяет для выбранного элемента конфигурировать все свойства, связанные со штрихами, с использованием окна `Properties`.

Применение графических трансформаций

В завершение обсуждения фигур будет рассмотрена тема *трансформаций*. Инфраструктура WPF поставляется с многочисленными классами, которые расширяют абстрактный базовый класс `System.Windows.Media.Transform`. В табл. 26.5 кратко описаны основные классы, производные от `Transform`.

Трансформации могут применяться к любым объектам `UIElement` (например, к объектам производных от `Shape` классов, а также к элементам управления `Button`, `TextBox` и т.п.). Используя классы трансформаций, можно визуализировать графические данные под заданным углом, скашивать изображение на поверхности и растягивать, сжимать либо поворачивать целевой элемент разными способами.

На заметку! Хотя объекты трансформаций могут применяться повсеместно, вы счтете их наиболее удобными при работе с анимацией WPF и специальными шаблонами элементов управления. Как будет показано далее в главе, анимацию WPF можно использовать для включения в специальный элемент управления визуальных подсказок, предназначенных конечному пользователю.

Таблица 26.5. Основные классы, производные от System.Windows.Media.Transform

Класс	Описание
MatrixTransform	Создает произвольную матричную трансформацию, которая используется для манипулирования объектами или координатными системами на двумерной плоскости
RotateTransform	Поворачивает объект по часовой стрелке вокруг указанной точки в двумерной системе координат (x, y)
ScaleTransform	Масштабирует объект в двумерной системе координат (x, y)
SkewTransform	Производит скашивание объекта в двумерной системе координат (x, y)
TranslateTransform	Преобразует (перемещает) объект в двумерной системе координат (x, y)
TransformGroup	Представляет комбинированный объект Transform, состоящий из других объектов Transform

Назначать целевому объекту (Button, Path и т.д.) трансформацию (либо целый набор трансформаций) можно с помощью двух общих свойств, LayoutTransform и RenderTransform.

Свойство LayoutTransform удобно тем, что трансформация происходит *перед* визуализацией элементов в диспетчере компоновки и потому не влияет на операции Z-упорядочивания (т.е. трансформируемые данные изображений не перекрываются).

С другой стороны, трансформация из свойства RenderTransform инициируется после того, как элементы попали в свои контейнеры, поэтому вполне возможно, что элементы будут трансформированы с перекрытием друг друга в зависимости от того, как они организованы в контейнере.

Первый взгляд на трансформации

Вскоре вы добавите к проекту RenderingWithShapes некоторую трансформирующую логику. Чтобы увидеть объект трансформации в действии, откройте редактор XAML, определите внутри корневого элемента Page или Window простой элемент StackPanel и установите свойство Orientation в Horizontal. Далее добавьте следующий элемент Rectangle, который будет нарисован под углом в 45 градусов с применением объекта RotateTransform:

```
<!-- Элемент Rectangle с трансформацией поворотом -->
<Rectangle Height ="100" Width ="40" Fill ="Red">
    <Rectangle.LayoutTransform>
        <RotateTransform Angle ="45"/>
    </Rectangle.LayoutTransform>
</Rectangle>
```

Здесь элемент Button скашивается на поверхности на 20 градусов посредством трансформации SkewTransform:

```
<!-- Элемент Button с трансформацией скашиванием -->
<Button Content ="Click Me!" Width="95" Height="40">
    <Button.LayoutTransform>
        <SkewTransform AngleX ="20" AngleY ="20"/>
    </Button.LayoutTransform>
</Button>
```

Для полноты картины ниже приведен элемент Ellipse, масштабированный на 20% с помощью трансформации ScaleTransform (обратите внимание на значения, установленные в свойствах Height и Width), а также элемент TextBox, к которому применена группа объектов трансформации:

```
<!-- Элемент Ellipse, масштабированный на 20% -->
<Ellipse Fill ="Blue" Width="5" Height="5">
    <Ellipse.LayoutTransform>
        <ScaleTransform ScaleX ="20" ScaleY ="20"/>
    </Ellipse.LayoutTransform>
</Ellipse>
<!-- Элемент TextBox, повернутый и скошенный -->
<TextBox Text ="Me Too!" Width="50" Height="40">
    <TextBox.LayoutTransform>
        <TransformGroup>
            <RotateTransform Angle ="45"/>
            <SkewTransform AngleX ="5" AngleY ="20"/>
        </TransformGroup>
    </TextBox.LayoutTransform>
</TextBox>
```

Следует отметить, что в случае применения трансформации выполнять какие-либо ручные вычисления для реагирования на проверку попадания, перемещение фокуса ввода и аналогичные действия не придется. Графический механизм WPF самостоятельно решает такие задачи. Например, на рис. 26.8 можно видеть, что элемент TextBox по-прежнему реагирует на клавиатурный ввод.



Рис. 26.8. Результат применения объектов графических трансформаций

Трансформация данных Canvas

Теперь нужно внедрить в пример RenderingWithShapes логику трансформации. Помимо применения объектов трансформации к одиночному элементу (Rectangle, TextBox и т.д.) их можно также применять к диспетчеру компоновки, чтобы трансформировать все внутренние данные. Например, всю панель DockPanel главного окна можно было бы визуализировать под углом:

```
<DockPanel LastChildFill="True">
    <DockPanel.LayoutTransform>
        <RotateTransform Angle="45"/>
    </DockPanel.LayoutTransform>
    ...
</DockPanel>
```

В рассматриваемом примере это несколько чрезмерно, так что добавьте последнюю (менее радикальную) возможность, которая позволит пользователю зеркально отобра-

зить целый контейнер Canvas и всю содержащуюся в нем графику. Начните с добавления в ToolBar финального элемента ToggleButton со следующим определением:

```
<ToggleButton Name="flipCanvas" Click="FlipCanvas_Click"
    Content="Flip Canvas!"/>
```

Внутри обработчика события Click для нового элемента ToggleButton создайте объект RotateTransform и подключите его к объекту Canvas через свойство LayoutTransform, если элемент ToggleButton отмечен. Если же элемент ToggleButton не отмечен, тогда удалите трансформацию, установив свойство LayoutTransform в null.

```
private void FlipCanvas_Click(object sender, RoutedEventArgs e)
{
    if (flipCanvas.IsChecked == true)
    {
        RotateTransform rotate = new RotateTransform(-180);
        canvasDrawingArea.LayoutTransform = rotate;
    }
    else
    {
        canvasDrawingArea.LayoutTransform = null;
    }
}
```

Запустите приложение и добавьте несколько графических фигур в область Canvas, следя за тем, чтобы они находились впритык к ее краям. После щелчка на новой кнопке обнаружится, что фигуры выходят за границы Canvas (рис. 26.9). Причина в том, что не был определен прямоугольник отсечения.

Исправить проблему легко. Вместо того чтобы вручную писать сложную логику отсечения, просто установите свойство ClipToBounds элемента Canvas в true, предотвратив визуализацию дочерних элементов вне границ родительского элемента. После запуска приложения можно заметить, что графические данные больше не покидают границы отведенной области.

```
<Canvas ClipToBounds = "True" ... >
```

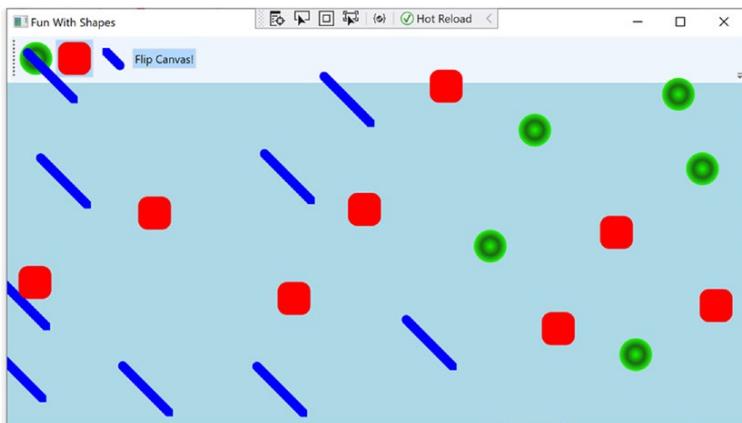


Рис. 26.9. После трансформации фигуры выходят за границы Canvas

Последняя крошка модификация, которую понадобится внести, связана с тем фактом, что когда пользователь зеркально отображает холст, щелкая на кнопке переключения, а затем щелкает на нем для рисования новой фигуры, то точка, где был произведен щелчок, не является той позицией, куда попадут графические данные. Взамен они появятся в месте нахождения курсора мыши.

Чтобы устранить проблему, примените тот же самый объект трансформации к рисуемой фигуре перед выполнением визуализации (через `RenderTransform`). Ниже показан основной фрагмент кода:

```
private void CanvasDrawingArea_MouseLeftButtonDown(object sender,
                                                    MouseButtonEventArgs e)
{
    // Для краткости код не показан.
    if (flipCanvas.IsChecked == true)
    {
        RotateTransform rotate = new RotateTransform(-180);
        shapeToRender.RenderTransform = rotate;
    }

    // Установить левую верхнюю точку для рисования на холсте.
    Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
    Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);

    // Нарисовать фигуру.
    canvasDrawingArea.Children.Add(shapeToRender);
}
```

На этом исследование пространства имен `System.Windows.Shapes`, кистей и трансформаций завершено. Прежде чем перейти к анализу роли визуализации графики с использованием рисунков и геометрических объектов, имеет смысл выяснить, каким образом IDE-среда Visual Studio способна упростить работу с примитивными графическими элементами.

Работа с редактором трансформаций Visual Studio

В предыдущем примере разнообразные трансформации применялись за счет ручного ввода разметки и написания кода C#. Наряду с тем, что поступать так вполне удобно, последняя версия Visual Studio поставляется со встроенным редактором трансформаций. Вспомните, что получателем служб трансформаций может быть любой элемент пользовательского интерфейса, в том числе диспетчер компоновки, содержащий различные элементы управления. Для демонстрации работы с редактором трансформаций Visual Studio будет создан новый проект приложения WPF по имени `FunWithTransforms`.

Построение начальной компоновки

Первым делом разделите первоначальный элемент `Grid` на две колонки с применением встроенного редактора сетки (точные размеры колонок роли не играют). Далее отыщите в панели инструментов элемент управления `StackPanel` и добавьте его так, чтобы он занял все пространство первой колонки `Grid`; затем добавьте в панель `StackPanel` три элемента управления `Button`:

```

<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <StackPanel Grid.Row="0" Grid.Column="0">
        <Button Name="btnSkew" Content="Skew" Click="Skew"/>
        <Button Name="btnRotate" Content="Rotate" Click="Rotate"/>
        <Button Name="btnFlip" Content="Flip" Click="Flip"/>
    </StackPanel>
</Grid>

```

Добавьте обработчики событий для кнопок:

```

private void Skew(object sender, RoutedEventArgs e)
{
}

private void Rotate(object sender, RoutedEventArgs e)
{
}

private void Flip(object sender, RoutedEventArgs e)
{
}

```

Чтобы завершить пользовательский интерфейс, создайте во второй колонке элемента Grid произвольную графику (используя любой прием, представленный ранее в главе). Вот разметка, применяемая в данном примере:

```

<Canvas x:Name="myCanvas" Grid.Column="1" Grid.Row="0">
    <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
        Height="186" Width="92" Stroke="Black"
        Canvas.Left="20" Canvas.Top="31">
        <Ellipse.Fill>
            <RadialGradientBrush>
                <GradientStop Color="#FF951ED8" Offset="0.215"/>
                <GradientStop Color="#FF2FECB0" Offset="1"/>
            </RadialGradientBrush>
        </Ellipse.Fill>
    </Ellipse>

    <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top"
        Height="101" Width="110" Stroke="Black"
        Canvas.Left="122" Canvas.Top="126">
        <Ellipse.Fill>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <GradientStop Color="#FFB91DDC" Offset="0.355"/>
                <GradientStop Color="#FFB0381D" Offset="1"/>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
</Canvas>

```

Окончательная компоновка показана на рис. 26.10.

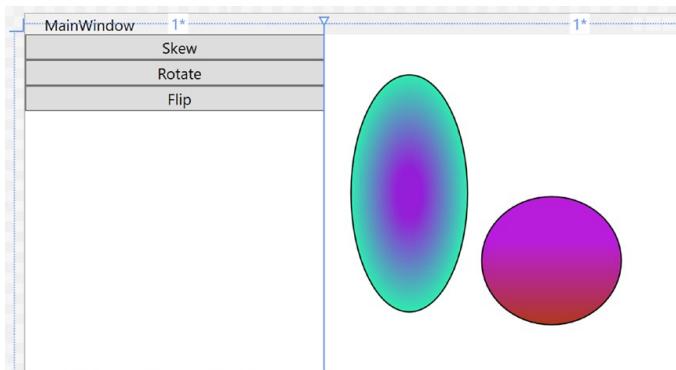


Рис. 26.10. Окончательная компоновка в примере трансформации

Применение трансформаций на этапе проектирования

Как упоминалось ранее, IDE-среда Visual Studio предоставляет встроенный редактор трансформаций, который можно найти в окне Properties. Раскройте раздел Transform (Трансформация), чтобы отобразить области RenderTransform и LayoutTransform редактора (рис. 26.11).

Подобно разделу Brush раздел Transform предлагает несколько вкладок, предназначенных для конфигурирования разнообразных типов графической трансформации текущего выбранного элемента. В табл. 26.6 описаны варианты трансформации, доступные на этих вкладках (в порядке слева направо).

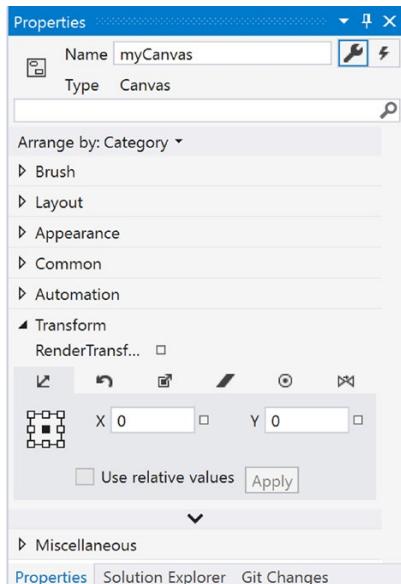


Рис. 26.11. Редактор трансформаций

Таблица 26.6. Варианты трансформации

Трансформация	Описание
Translate (Трансляция)	Позволяет сдвинуть местоположение элемента в позицию (x, y)
Rotate (Поворот)	Позволяет повернуть элемент на угол до 360 градусов
Scale (Масштабирование)	Позволяет увеличить или уменьшить элемент на масштабный коэффициент в направлениях x и y
Skew (Перекашивание)	Позволяет скосить ограничивающий прямоугольник, содержащий выбранный элемент, на масштабный коэффициент в направлениях x и y
Center Point (Центральная точка)	При повороте или зеркальном отображении объекта элемент перемещается относительно фиксированной точки, которая называется центральной точкой объекта. По умолчанию центральная точка объекта расположена в центре объекта; тем не менее, эта трансформация позволяет изменять центральную точку объекта для выполнения поворота или зеркального отображения относительно другой точки
Flip (Зеркальное отображение)	Позволяет зеркально отобразить выбранный элемент на основе координаты x или y центральной точки

Испытайте каждую из описанных трансформаций, используя в качестве цели специальную фигуру (для отмены выполненной операции просто нажмите <Ctrl+Z>). Как и многие другие аспекты раздела Transform окна Properties, каждая трансформация имеет уникальный набор параметров конфигурации, которые должны стать вполне понятными, как только вы просмотрите их. Например, редактор трансформации Skew позволяет устанавливать значения скоса x и y, а редактор трансформации Flip дает возможность зеркально отображать относительно оси x или y и т.д.

Трансформация холста в коде

Реализации обработчиков для всех кнопок будут более или менее похожими. Мы сконфигурируем объект трансформации и присвоим его объекту myCanvas. Затем после запуска приложения можно будет щелкать на кнопке, чтобы просматривать результат применения трансформации. Ниже приведен полный код обработчиков (обратите внимание на установку свойства LayoutTransform, что позволяет фигурам позиционироваться относительно родительского контейнера):

```
private void Flip(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}

private void Rotate(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new RotateTransform(180);
}

private void Skew(object sender, System.Windows.RoutedEventArgs e)
{
    myCanvas.LayoutTransform = new SkewTransform(40, -20);
}
```

Визуализация графических данных с использованием рисунков и геометрических объектов

Несмотря на то что типы Shape позволяют генерировать интерактивную двумерную поверхность любого вида, из-за насыщенной цепочки наследования они потребляют довольно много памяти. И хотя класс Path может помочь снизить накладные расходы за счет применения включенных геометрических объектов (вместо крупной коллекции других фигур), инфраструктура WPF предоставляет развитый API-интерфейс рисования и геометрии, который визуализирует еще более легковесные двумерные векторные изображения.

Входной точкой в этот API-интерфейс является абстрактный класс `System.Windows.Media.Drawing` (из сборки `PresentationCore.dll`), который сам по себе всего лишь определяет ограничивающий прямоугольник для хранения результатов визуализации.

Инфраструктура WPF предлагает разнообразные классы, расширяющие `Drawing`, каждый из которых представляет отдельный способ рисования содержимого (табл. 26.7).

Таблица 26.7. Классы, производные от Drawing

Класс	Описание
<code>DrawingGroup</code>	Используется для комбинирования коллекции отдельных объектов, производных от <code>Drawing</code> , в единую составную визуализацию
<code>GeometryDrawing</code>	Применяется для визуализации двумерных фигур в очень легковесной манере
<code>GlyphRunDrawing</code>	Используется для визуализации текстовых данных с применением служб графической визуализации WPF
<code>ImageDrawing</code>	Используется для визуализации файла изображения, или набора геометрических объектов, внутри ограничивающего прямоугольника
<code>VideoDrawing</code>	Применяется для воспроизведения аудио- или видеоролика. Этот тип может полноценно использоваться только в процедурном коде. Для воспроизведения видеоролика в разметке XAML гораздо лучше подходит тип <code>MediaPlayer</code>

Будучи более легковесными, производные от `Drawing` типы не обладают встроенной возможностью обработки событий, т.к. они не являются `UIElement` или `FrameworkElement` (хотя допускают программную реализацию логики проверки попадания).

Другое ключевое отличие между типами, производными от `Drawing`, и типами, производными от `Shape`, состоит в том, что производные от `Drawing` типы не умеют визуализировать себя, поскольку не унаследованы от `UIElement`! Для отображения содержимого производные типы должны помещаться в какой-то контейнерный объект (в частности `DrawingImage`, `DrawingBrush` или `DrawingVisual`).

Класс `DrawingImage` позволяет помещать рисунки и геометрические объекты внутрь элемента управления `Image` из WPF, который обычно применяется для отображения данных из внешнего файла. Класс `DrawingBrush` дает возможность строить кисть на основе рисунков и геометрических объектов, которая предназначена для ус-

тановки свойства, требующего кисть. Наконец, класс DrawingVisual используется только на “визуальном” уровне графической визуализации, полностью управляемом из кода C#.

Хотя работать с рисунками немного сложнее, чем с простыми фигурами, отделение графической композиции от графической визуализации делает типы, производные от Drawing, гораздо более легковесными, чем производные от Shape типы, одновременно сохраняя их ключевые службы.

Построение кисти DrawingBrush с использованием геометрических объектов

Ранее в главе элемент Path заполнялся группой геометрических объектов примерно так:

```
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
    <Path.Data>
        <GeometryGroup>
            <EllipseGeometry Center = "75,70" RadiusX = "30" RadiusY = "30" />
            <RectangleGeometry Rect = "25,55 100 30" />
            <LineGeometry StartPoint="0,0" EndPoint="70,30" />
            <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

Поступая подобным образом, вы достигаете интерактивности Path при чрезвычайной легковесности, присущей геометрическим объектам. Однако если необходимо визуализировать аналогичный вывод и отсутствует потребность в любой (готовой) интерактивности, тогда тот же самый элемент <GeometryGroup> можно поместить внутрь DrawingBrush:

```
<DrawingBrush>
    <DrawingBrush.Drawing>
        <GeometryDrawing>
            <GeometryDrawing.Geometry>
                <GeometryGroup>
                    <EllipseGeometry Center = "75,70"
                        RadiusX = "30" RadiusY = "30" />
                    <RectangleGeometry Rect = "25,55 100 30" />
                    <LineGeometry StartPoint="0,0" EndPoint="70,30" />
                    <LineGeometry StartPoint="70,30" EndPoint="0,30" />
                </GeometryGroup>
            </GeometryDrawing.Geometry>
            <!-- Специальное перо для рисования границ -->
            <GeometryDrawing.Pen>
                <Pen Brush="Blue" Thickness="3"/>
            </GeometryDrawing.Pen>
            <!-- Специальная кисть для заполнения внутренней области -->
            <GeometryDrawing.Brush>
                <SolidColorBrush Color="Orange"/>
            </GeometryDrawing.Brush>
        </GeometryDrawing>
    </DrawingBrush.Drawing>
</DrawingBrush>
```

При помещении группы геометрических объектов внутрь DrawingBrush также понадобится установить объект Pen, применяемый для рисования границ, потому что свойство Stroke больше не наследуется от базового класса Shape. Здесь был создан элемент Pen с теми же настройками, которые использовались в значениях Stroke и StrokeThickness из предыдущего примера Path.

Кроме того, поскольку свойство Fill больше не наследуется от класса Shape, нужно также применять синтаксис “элемент-свойство” для определения объекта кисти, предназначенного элементу DrawingGeometry, со сплошным оранжевым цветом, как в предыдущих настройках Path.

Рисование с помощью DrawingBrush

Теперь объект DrawingBrush можно использовать для установки значения любого свойства, требующего объекта кисти. Например, после подготовки следующей разметки в редакторе XAML с помощью синтаксиса “элемент-свойство” можно рисовать изображение по всей поверхности Page:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Page.Background>
        <DrawingBrush>
            <!-- Тот же самый объект DrawingBrush, что и ранее -->
        </DrawingBrush>
    </Page.Background>
</Page>
```

Или же элемент DrawingBrush можно применять для установки другого совместимого с кистью свойства, такого как свойство Background элемента Button:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Button Height="100" Width="100">
        <Button.Background>
            <DrawingBrush>
                <!-- Тот же самый объект DrawingBrush, что и ранее -->
            </DrawingBrush>
        </Button.Background>
    </Button>
</Page>
```

Независимо от того, какое совместимое с кистью свойство устанавливается с использованием специального объекта DrawingBrush, визуализация двумерного графического изображения в итоге получается с намного меньшими накладными расходами, чем в случае визуализации того же изображения посредством фигур.

Включение типов Drawing в DrawingImage

Тип DrawingImage позволяет подключать рисованный геометрический объект к элементу управления Image из WPF. Взгляните на следующую разметку:

```
<Image>
    <Image.Source>
        <DrawingImage>
```

```

<DrawingImage.Drawing>
    <!-- Тот же самый объект DrawingBrush, что и ранее -->
    </DrawingImage.Drawing>
</DrawingImage>
</Image.Source>
</Image>

```

В данном случае элемент GeometryDrawing был помещен внутрь элемента DrawingImage, а не DrawingBrush. С применением элемента DrawingImage можно установить свойство Source элемента управления Image.

Работа с векторными изображениями

По всей видимости, вы согласитесь с тем, что художнику будет довольно трудно создавать сложное векторное изображение с использованием инструментов и приемов, предоставляемых средой Visual Studio. В распоряжении художников есть собственные наборы инструментов, которые позволяют производить замечательную векторную графику. Изобразительными возможностями подобного рода не обладает ни IDE-среда Visual Studio, ни сопровождающий ее инструмент Microsoft Blend. Перед тем, как векторные изображения можно будет импортировать в приложение WPF, они должны быть преобразованы в выражения путей. После этого можно программировать с применением генерированной объектной модели, используя Visual Studio.

На заметку! Используемое изображение (LaserSign.svg) и экспортированные данные путей (LaserSign.xaml) можно найти в подкаталоге Chapter_26 загружаемого кода примеров. Изображение взято из статьи Википедии по ссылке https://ru.wikipedia.org/wiki/Символы_опасности.

Преобразование файла с векторной графикой в файл XAML

Прежде чем можно будет импортировать сложные графические данные (такие как векторная графика) в приложение WPF, графику понадобится преобразовать в данные путей. Чтобы проиллюстрировать, как это делается, возьмите пример файла изображения .svg с упомянутым выше знаком опасности лазерного излучения. Затем загрузите и установите инструмент с открытым кодом под названием Inkscape (из веб-сайта www.inkscape.org). С помощью Inkscape откройте файл LaserSign.svg из подкаталога Chapter_26. Вы можете получить запрос о модернизации формата. Установите настройки, как показано на рис. 26.12.

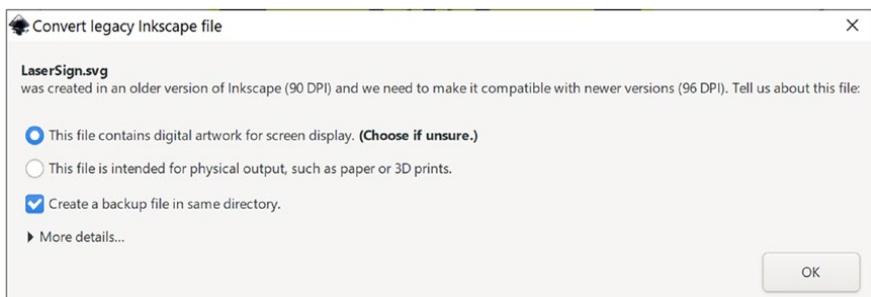


Рис. 26.12. Модернизация файла SVG до самого последнего формата в Inkscape

Следующие шаги поначалу покажутся несколько странными, но на самом деле они представляют собой простой способ преобразования векторных изображений в разметку XAML. Когда изображение приобрело желаемый вид, необходимо выбрать пункт меню File⇒Print (Файл⇒Print (Печать)). В открывшемся окне нужно ввести имя файла и выбрать место, где он должен быть сохранен, после чего щелкнуть на кнопке Save (Сохранить). В результате получается файл *.xps (или *.oxps).

На заметку! В зависимости от нескольких переменных среды в конфигурации системы сгенерированный файл будет иметь либо расширение .xps, либо расширение .oxps. В любом случае дальнейший процесс идентичен.

Форматы *.xps и *.oxps в действительности представляют собой архивы ZIP. Переименовав расширение в .zip, файл можно открыть в проводнике файлов (либо в 7-Zip или в предпочтаемой утилите архивации). Файл содержит иерархию папок, приведенную на рис. 26.13.

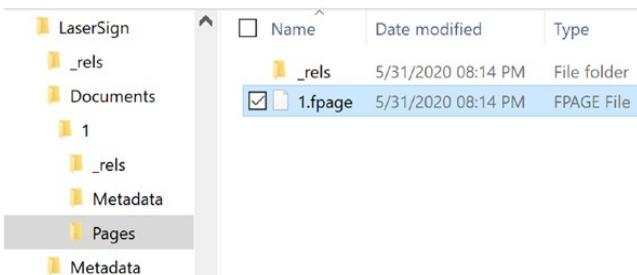


Рис. 26.13. Иерархия папок в файле *.xps или *.oxps

Необходимый файл находится в папке Pages (Documents/1/Pages) и называется 1.fpage. Откройте его в текстовом редакторе и скопируйте в буфер все данные кроме открывающего и закрывающего дескрипторов FixedPage. Данные путей затем можно поместить внутрь элемента Canvas главного окна в XAML. В итоге изображение будет показано в окне XAML.

На заметку! В последней версии Inkscape есть возможность сохранить файл в формате Microsoft XAML. К сожалению, на момент написания главы он не был совместим с WPF.

Импортирование графических данных в проект WPF

Создайте новый проект приложения WPF по имени InteractiveLaserSign. Измените значения свойств Height и Width элемента Window соответственно на 600 и 650 и замените элемент Grid элементом Canvas:

```
<Window x:Class="InteractiveLaserSign.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:InteractiveLaserSign"
    mc:Ignorable="d">
```

```
Title="MainWindow" Height="600" Width="650">
<Canvas>
</Canvas>
</Window>
```

Скопируйте полную разметку XAML из файла 1.xaml (исключая внешний дескриптор FixedPage) и вставьте ее в элемент управления Canvas внутри MainWindow. Просмотрев окно в режиме проектирования, легко удостовериться в том, что знак опасности лазерного излучения успешно воспроизводится в приложении.

Заглянув в окно Document Outline, вы заметите, что каждая часть изображения представлена как XAML-элемент Path. Если изменить размеры элемента Window, то качество изображения останется тем же самым безотносительно к тому, насколько большим сделано окно. Причина в том, что изображения, представленные с помощью элементов Path, визуализируются с применением механизма рисования и математики, а не за счет манипулирования пикселями.

Взаимодействие с изображением

Вспомните, что маршрутизируемое событие распространяется туннельным и пузырьковым образом, поэтому щелчок на любом элементе Path внутри Canvas может быть обработан обработчиком событий щелчка на Canvas. Модифицируйте разметку Canvas следующим образом:

```
<Canvas MouseLeftButtonDown="Canvas_MouseLeftButtonDown">
```

Добавьте обработчик событий с таким кодом:

```
private void Canvas_MouseLeftButtonDown(object sender,
                                         MouseButtonEventArgs e)
{
    if (e.OriginalSource is Path p)
    {
        p.Fill = new SolidColorBrush(Colors.Red);
    }
}
```

Запустите приложение и щелкните на линиях, чтобы увидеть эффекты.

Теперь вы понимаете процесс генерации данных путей для сложной графики и знаете, как взаимодействовать с графическими данными в коде. Вы наверняка согласитесь, что наличие у профессиональных художников возможности генерировать сложные графические данные и экспорттировать их в виде разметки XAML исключительно важна. После того как графические данные сохранены в файле XAML, разработчики могут импортировать разметку и писать код для взаимодействия с объектной моделью.

Визуализация графических данных с использованием визуального уровня

Последний вариант визуализации графических данных с помощью WPF называется *визуальным уровнем*. Ранее уже упоминалось, что доступ к нему возможен только из кода (он не дружественен по отношению к разметке XAML). Несмотря на то что подавляющее большинство приложений WPF будут хорошо работать с применением фигур, рисунков и геометрических объектов, визуальный уровень обеспечивает самый

быстрый способ визуализации крупных объемов графических данных. Визуальный уровень также может быть полезен, когда необходимо визуализировать единственное изображение в крупной области. Например, если требуется заполнить фон окна простым статическим изображением, тогда визуальный уровень будет наиболее быстрым способом решения такой задачи. Кроме того, он удобен, когда нужно очень быстро менять фон окна в зависимости от ввода пользователя или чего-нибудь еще.

Далее будет построена небольшая программа, иллюстрирующая основы использования визуального уровня.

Базовый класс **Visual** и производные дочерние классы

Абстрактный класс `System.Windows.Media.Visual` предлагает минимальный набор служб (визуализацию, проверку попадания, трансформации) для визуализации графики, но не предоставляет поддержку дополнительных невизуальных служб, которые могут приводить к разбуханию кода (события ввода, службы компоновки, стили и привязка данных). Класс `Visual` является абстрактным базовым классом. Для выполнения действительных операций визуализации должен применяться один из его производных классов. В WPF определено несколько подклассов `Visual`, в том числе `DrawingVisual`, `Viewport3DVisual` и `ContainerVisual`.

Рассматриваемый ниже пример сосредоточен только на `DrawingVisual` — легковесном классе рисования, который используется для визуализации фигур, изображений или текста.

Первый взгляд на класс **DrawingVisual**

Чтобы визуализировать данные на поверхности с применением класса `DrawingVisual`, понадобится выполнить следующие основные шаги:

- получить объект `DrawingContext` из `DrawingVisual`;
- использовать объект `DrawingContext` для визуализации графических данных.

Эти два шага представляют абсолютный минимум, необходимый для визуализации каких-то данных на поверхности. Тем не менее, когда нужно, чтобы визуализируемые графические данные реагировали на вычисления при проверке попадания (что важно для добавления взаимодействия с пользователем), потребуется также выполнить дополнительные шаги:

- обновить логическое и визуальное деревья, поддерживаемые контейнером, на котором производится визуализация;
- переопределить два виртуальных метода из класса `FrameworkElement`, позволяя контейнеру получать созданные визуальные данные.

Давайте исследуем последние два шага более подробно. Чтобы продемонстрировать применение класса `DrawingVisual` для визуализации двумерных данных, создайте в Visual Studio новый проект приложения WPF по имени `RenderingWithVisuals`. Первой целью будет использование класса `DrawingVisual` для динамического присваивания данных элементу управления `Image` из WPF. Начните со следующего обновления разметки XAML окна для обработки события `Loaded`:

```
<Window x:Class="RenderingWithVisuals.MainWindow"
       <!-- Для краткости разметка не показана -->
       Title="Fun With Visual Layer" Height="450" Width="800"
       Loaded="MainWindow_Loaded">
```

Замените элемент Grid панелью StackPanel и добавьте в нее элемент Image:

```
<StackPanel Background="AliceBlue" Name="myStackPanel">
    <Image Name="myImage" Height="80"/>
</StackPanel>
```

Элемент управления Image пока не имеет значения в свойстве Source, т.к. оно будет устанавливаться во время выполнения. С событием Loaded связана работа по построению графических данных в памяти с применением объекта DrawingBrush. Удостоверьтесь в том, что файл MainWindow.cs содержит операторы using для следующих пространств имен:

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;
```

Вот реализация обработчика события Loaded:

```
private void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    const int TextFontSize = 30;

    // Создать объект System.Windows.Media.FormattedText.
    FormattedText text = new FormattedText("Hello Visual Layer!",
        new System.Globalization.CultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface(this.FontFamily, FontStyles.Italic,
            FontWeights.DemiBold, FontStretches.UltraExpanded),
        TextFontSize,
        Brushes.Green,
        null,
        VisualTreeHelper.GetDpi(this).PixelsPerDip);

    // Создать объект DrawingVisual и получить объект DrawingContext.
    DrawingVisual drawingVisual = new DrawingVisual();
    using(DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
        // Вызвать любой из методов DrawingContext для визуализации данных.
        drawingContext.DrawRoundedRectangle(Brushes.Yellow,
            new Pen(Brushes.Black, 5),
            new Rect(5, 5, 450, 100), 20, 20);
        drawingContext.DrawText(text, new Point(20, 20));
    }

    // Динамически создать битовое изображение,
    // используя данные в объекте DrawingVisual.
    RenderTargetBitmap bmp = new RenderTargetBitmap(500, 100, 100, 90,
        PixelFormats.Pbgra32);
    bmp.Render(drawingVisual);

    // Установить источник для элемента управления Image.
    myImage.Source = bmp;
}
```

В коде задействовано несколько новых классов WPF, которые будут кратко описаны ниже. Метод начинается с создания нового объекта FormattedText, который представляет текстовую часть конструируемого изображения в памяти.

Как видите, конструктор позволяет указывать многочисленные атрибуты, в том числе размер шрифта, семейство шрифтов, цвет переднего плана и сам текст.

Затем через вызов метода `RenderOpen()` на экземпляре `DrawingVisual` получается необходимый объект `DrawingContext`. Здесь в `DrawingVisual` визуализируется цветной прямоугольник со скругленными углами, за которым следует форматированный текст. В обоих случаях графические данные помещаются в `DrawingVisual` с применением жестко закодированных значений, что не слишком хорошо в производственном приложении, но вполне подходит для такого простого теста.

Несколько последних операторов отображают `DrawingVisual` на объект `RenderTargetBitmap`, который является членом пространства имен `System.Windows.Media.Imaging`. Этот класс принимает визуальный объект и трансформирует его в растровое изображение, находящееся в памяти. Затем устанавливается свойство `Source` элемента управления `Image` и получается вывод, показанный на рис. 26.14.

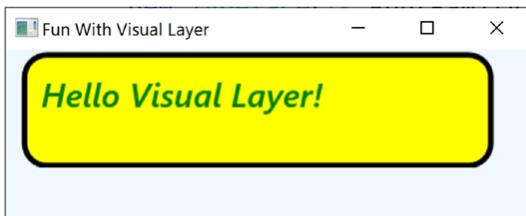


Рис. 26.14. Использование визуального уровня для визуализации растрового изображения, расположенного в памяти

На заметку! Пространство имен `System.Windows.Media.Imaging` содержит дополнительные классы кодирования, которые позволяют сохранять находящийся в памяти объект `RenderTargetBitmap` в физический файл в разнообразных форматах. Детали ищите в описании `JpegBitmapEncoder` и связанных с ним классов.

Визуализация графических данных в специальном диспетчере компоновки

Хотя применение `DrawingVisual` для рисования на фоне элемента управления WPF представляет интерес, возможно чаще придется строить специальный диспетчер компоновки (`Grid`, `StackPanel`, `Canvas` и т.д.), который внутренне использует визуальный уровень для визуализации своего содержимого. После создания такого специального диспетчера компоновки его можно подключить к обычному элементу `Window` (а также `Page` или `UserControl`) и позволить части пользовательского интерфейса использовать высоко оптимизированный агент визуализации, в то время как для визуализации некритичных графических данных будут применяться фигуры и рисунки.

Если дополнительная функциональность, предлагаемая специализированным диспетчером компоновки, не требуется, то можно просто расширить класс `FrameworkElement`, который обладает необходимой инфраструктурой, позволяющей содержать также и визуальные элементы. В целях иллюстрации вставьте в проект новый класс по имени `CustomVisualFrameworkElement`.

Унаследуйте его от FrameworkElement и импортируйте пространства имен System, System.Windows, System.Windows.Input, System.Windows.Media и System.Windows.Media.Imaging.

Класс CustomVisualFrameworkElement будет поддерживать переменную-член типа VisualCollection, которая содержит два фиксированных объекта DrawingVisual (конечно, в эту коллекцию можно было бы добавлять члены с помощью мыши, но лучше сохранить пример простым). Модифицируйте код класса следующим образом:

```
public class CustomVisualFrameworkElement : FrameworkElement
{
    // Коллекция всех визуальных объектов.
    VisualCollection theVisuals;

    public CustomVisualFrameworkElement()
    {
        //Заполнить коллекцию VisualCollection несколькими объектами DrawingVisual.
        // Аргумент конструктора представляет владельца визуальных объектов.
        theVisuals = new VisualCollection(this)
        { AddRect(), AddCircle() };
    }

    private Visual AddCircle()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
        // Получить объект DrawingContext для создания нового содержимого.
        using (DrawingContext drawingContext =
            drawingVisual.RenderOpen())
        {
            // Создать круг и нарисовать его в DrawingContext.
            drawingContext.DrawEllipse(Brushes.DarkBlue, null,
                new Point(70, 90), 40, 50);
            return drawingVisual;
        }
    }

    private Visual AddRect()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
        using (DrawingContext drawingContext =
            drawingVisual.RenderOpen());
        Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
        drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
        return drawingVisual;
    }
}
```

Прежде чем специальный элемент FrameworkElement можно будет использовать внутри Window, потребуется переопределить два упомянутых ранее ключевых виртуальных члена, которые вызываются внутренне инфраструктурой WPF во время процесса визуализации. Метод GetVisualChild() возвращает из коллекции дочерних элементов дочерний элемент по указанному индексу. Свойство VisualChildrenCount, допускающее только чтение, возвращает количество визуальных дочерних элементов внутри визуальной коллекции. Оба члена легко реализовать, т.к. всю реальную работу можно делегировать переменной-члену типа VisualCollection:

```

protected override int VisualChildrenCount => theVisuals.Count;
protected override Visual GetVisualChild(int index)
{
    // Значение должно быть больше нуля, поэтому разумно это проверить.
    if (index < 0 || index >= theVisuals.Count)
    {
        throw new ArgumentOutOfRangeException();
    }
    return theVisuals[index];
}

```

Теперь вы располагаете достаточной функциональностью, чтобы протестировать специальный класс. Модифицируйте описание XAML элемента Window, добавив в существующий контейнер StackPanel один объект CustomVisualFrameworkElement. Это потребует создания специального пространства имен XML, которое отображается на пространство имен .NET Core.

```

<Window x:Class="RenderingWithVisuals.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:local="clr-namespace:RenderingWithVisuals"
       Title="Fun with the Visual Layer" Height="350" Width="525"
       Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
        <local:CustomVisualFrameworkElement/>
    </StackPanel>
</Window>

```

Результат выполнения программы показан на рис. 26.15.

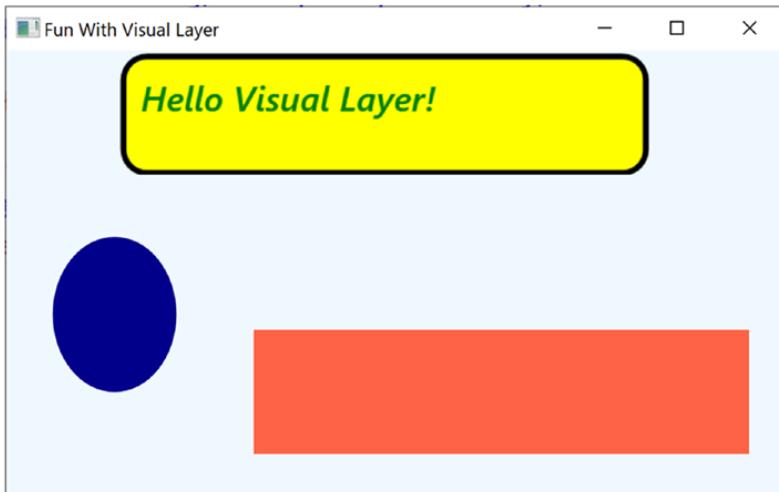


Рис. 26.15. Использование визуального уровня для визуализации данных в специальном элементе FrameworkElement

Реагирование на операции проверки попадания

Поскольку класс DrawingVisual не располагает инфраструктурой UIElement или FrameworkElement, необходимо программно добавить возможность реагирования на операции проверки попадания. Благодаря концепции логического и визуального дерева на визуальном уровне делать это очень просто. Оказывается, что в результате написания блока XAML по существу строится логическое дерево элементов. Однако с каждым логическим деревом связано намного более развитое описание, известное как визуальное дерево, которое содержит низкоуровневые инструкции визуализации.

Упомянутые деревья подробно рассматриваются в главе 27, а сейчас достаточно знать, что до тех пор, пока специальные визуальные объекты не будут зарегистрированы в таких структурах данных, выполнять операции проверки попадания невозможно. К счастью, контейнер VisualCollection обеспечивает регистрацию автоматически (вот почему в аргументе конструктора необходимо передавать ссылку на специальный элемент FrameworkElement).

Измените код класса CustomVisualFrameworkElement для обработки события MouseDown в конструкторе класса с применением стандартного синтаксиса C#:

```
this.MouseDown += CustomVisualFrameworkElement_MouseDown;
```

Реализация данного обработчика будет вызывать метод VisualTreeHelper.HitTest() с целью выяснения, находится ли курсор мыши внутри границ одного из визуальных объектов. Для этого в одном из параметров метода HitTest() указывается делегат HitTestResultCallback, который будет выполнять вычисления. Добавьте в класс CustomVisualFrameworkElement следующие методы:

```
void CustomVisualFrameworkElement_MouseDown(object sender,
                                              MouseButtonEventArgs e)
{
    // Выяснить, где пользователь выполнил щелчок.
    Point pt = e.GetPosition((UIElement)sender);

    // Вызвать вспомогательную функцию через делегат, чтобы
    // посмотреть, был ли совершен щелчок на визуальном объекте.
    VisualTreeHelper.HitTest(this, null,
        new HitTestResultCallback(myCallback),
        new PointHitTestParameters(pt));
}

public HitTestResultBehavior myCallback(HitTestResult result)
{
    // Если щелчок был совершен на визуальном объекте, то
    // переключиться между сконченной и нормальной визуализацией.
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (((DrawingVisual)result.VisualHit).Transform == null)
        {
            ((DrawingVisual)result.VisualHit).Transform =
                new SkewTransform(7, 7);
        }
        else
        {
            ((DrawingVisual)result.VisualHit).Transform = null;
        }
    }
}
```

```
// Сообщить методу HitTest() о прекращении углубления в визуальное дерево
return HitTestResultBehavior.Stop;
}
```

Снова запустите программу. Теперь должна появиться возможность щелкать на любом из отображенных визуальных объектов и наблюдать за выполнением трансформации. Наряду с тем, что рассмотренный пример взаимодействия с визуальным уровнем WPF очень прост, не забывайте, что здесь можно использовать те же самые кисти, трансформации, перья и диспетчеры компоновки, которые обычно применяются в разметке XAML. Таким образом, вы уже знаете довольно много о работе с классами, производными от *Visual*.

На этом исследование служб графической визуализации WPF завершено. Несмотря на раскрытие ряда интересных тем, на самом деле мы лишь слегка затронули обширную область графических возможностей инфраструктуры WPF. Дальнейшее изучение фигур, рисунков, кистей, трансформаций и визуальных объектов вы можете продолжить самостоятельно (в оставшихся главах, посвященных WPF, еще встретятся дополнительные детали).

Резюме

Поскольку Windows Presentation Foundation является настолько насыщенной графикой инфраструктурой для построения графических пользовательских интерфейсов, не должно вызывать удивления наличие нескольких способов визуализации графического вывода. Глава начиналась с рассмотрения трех подходов к визуализации (фигуры, рисунки и визуальные объекты), а также разнообразных примитивов визуализации, таких как кисти, перья и трансформации.

Вспомните, что когда необходимо строить интерактивную двумерную визуализацию, то фигуры делают такой процесс очень простым. С другой стороны, статические, не интерактивные изображения могут визуализироваться в оптимальной манере с использованием рисунков и геометрических объектов, а визуальный уровень (доступный только в коде) обеспечит максимальный контроль и производительность.

ГЛАВА 27

Ресурсы, анимация, стили и шаблоны WPF

В настоящей главе будут представлены три важные (и взаимосвязанные) темы, которые позволят углубить понимание API-интерфейса Windows Presentation Foundation (WPF). Первым делом вы изучите роль логических ресурсов. Вы увидите, что система логических ресурсов (также называемых *объектными ресурсами*) представляет собой способ ссылки на часто используемые объекты внутри приложения WPF. Хотя логические ресурсы нередко реализуются в разметке XAML, они могут быть определены и в процедурном коде.

Далее вы узнаете, как определять, выполнять и управлять анимационной последовательностью. Вопреки тому, что можно было подумать, применение анимации WPF не ограничивается видеоиграми или мультимедийными приложениями. В API-интерфейсе WPF анимация может использоваться, например, для подсветки кнопки, когда она получает фокус, или увеличения размера выбранной строки в DataGrid. Понимание анимации является ключевым аспектом построения специальных шаблонов элементов управления (как вы увидите позже в главе).

Затем объясняется роль стилей и шаблонов WPF. Подобно веб-странице, в которой применяются стили CSS или механизм тем ASP.NET, приложение WPF может определять общий вид и поведение для набора элементов управления. Такие стили можно определять в разметке и сохранять их в виде объектных ресурсов для последующего использования, а также динамически применять во время выполнения. В последнем примере вы научитесь строить специальные шаблоны элементов управления.

Система ресурсов WPF

Первой задачей будет исследование темы встраивания и доступа к ресурсам приложения. Инфраструктура WPF поддерживает два вида ресурсов. Первый из них — *двоичные ресурсы*; эта категория обычно включает элементы, которые большинство программистов считают ресурсами в традиционном смысле (встроенные файлы изображений или звуковых клипов, значки, используемые приложением, и т.д.).

Вторая категория, называемая *объектными ресурсами* или *логическими ресурсами*, представляет именованные объекты .NET, которые можно упаковывать и многократно применять повсюду в приложении. Несмотря на то что упаковывать в виде объектного ресурса разрешено любой объект .NET, логические ресурсы особенно удобны при работе с графическими данными произвольного рода, поскольку можно определить часто используемые графические примитивы (кисти, перья, анимации и т.д.) и ссылаться на них по мере необходимости.

Работа с двоичными ресурсами

Прежде чем перейти к теме объектных ресурсов, давайте кратко проанализируем, как упаковывать *двоичные ресурсы* вроде значков и файлов изображений (например, логотипов компаний либо изображений для анимации) внутри приложений. Создайте в Visual Studio новый проект приложения WPF по имени BinaryResourcesApp. Модифицируйте разметку начального окна для обработки события Loaded элемента Window и применения DockPanel в качестве корня компоновки:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
    <!-- Для краткости разметка не показана -->
    Title="Fun with Binary Resources" Height="500" Width="649"
    Loaded="MainWindow_OnLoaded">
    <DockPanel LastChildFill="True">
    </DockPanel>
</Window>
```

Предположим, что приложение должно отображать внутри части окна один из трех файлов изображений, основываясь на пользовательском вводе. Элемент управления Image из WPF может использоваться для отображения не только типичного файла изображения (*.bmp, *.gif, *.ico, *.jpg, *.png, *.wdp или *.tiff), но также данных объекта DrawingImage (как было показано в главе 26). Можете построить пользовательский интерфейс окна, который поддерживает диспетчер компоновки DockPanel, содержащий простую панель инструментов с кнопками Next (Вперед) и Previous (Назад). Ниже панели инструментов расположите элемент управления Image, свойство Source которого в текущий момент не установлено:

```
<DockPanel LastChildFill="True">
    <ToolBar Height="60" Name="picturePickerToolbar" DockPanel.Dock="Top">
        <Button x:Name="btnPreviousImage" Height="40"
            Width="100" BorderBrush="Black"
            Margin="5" Content="Previous"
            Click="btnPreviousImage_Click"/>
        <Button x:Name="btnNextImage" Height="40"
            Width="100" BorderBrush="Black"
            Margin="5" Content="Next" Click="btnNextImage_Click"/>
    </ToolBar>
    <!-- Этот элемент Image будет заполняться в коде -->
    <Border BorderThickness="2" BorderBrush="Green">
        <Image x:Name="imageHolder" Stretch="Fill" />
    </Border>
</DockPanel>
```

Добавьте следующие пустые обработчики событий:

```
private void MainWindow_OnLoaded(
    object sender, RoutedEventArgs e)
{
}

private void btnPreviousImage_Click(
    object sender, RoutedEventArgs e)
{
}
```

```
private void btnNextImage_Click(
    object sender, RoutedEventArgs e)
{
}
```

Во время загрузки окна изображения добавляются в коллекцию, по которой будет совершаться проход с помощью кнопок Next и Previous. Располагая инфраструктурой приложения, можно заняться исследованием разных вариантов ее реализации.

Включение в проект несвязанных файлов ресурсов

Один из вариантов предусматривает поставку файлов изображений в виде набора несвязанных файлов в каком-то подкаталоге внутри пути установки приложения. Начните с создания в проекте новой папки (по имени Images). Добавьте в папку несколько изображений, щелкнув правой кнопкой мыши внутри данной папки и выбрав в контекстном меню пункт Add⇒Existing Item (Добавить⇒Существующий элемент). В открывшемся диалоговом окне Add Existing Item (Добавление существующего элемента) измените фильтр файлов на *.*., чтобы стали видны файлы изображений. Вы можете добавлять собственные файлы изображений или задействовать три файла изображений с именами Deer.jpg, Dogs.jpg и Welcome.jpg из загружаемого кода примеров.

Конфигурирование несвязанных ресурсов

Чтобы скопировать содержимое папки \Images в папку \bin\Debug при компиляции проекта, выберите все изображения в окне Solution Explorer, щелкните правой кнопкой мыши и выберите в контекстном меню пункт Properties (Свойства); откроется окно Properties (Свойства). Установите свойство Build Action (Действие сборки) в Content (Содержимое), а свойство Copy Output Directory (Копировать в выходной каталог) в Copy always (Копировать всегда), как показано на рис. 27.1.

На заметку! Для свойства Copy Output Directory можно было бы также выбрать вариант Copy if Newer (Копировать, если новее), что позволит сократить время копирования при построении крупных проектов с большим объемом содержимого. В рассматриваемом примере варианта Copy always вполне достаточно.

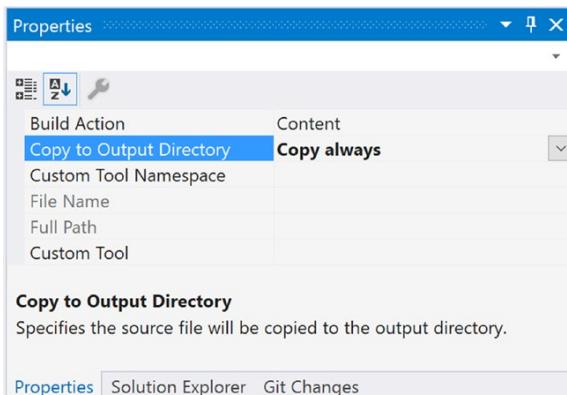


Рис. 27.1. Конфигурирование данных изображений для копирования в выходной каталог

После компиляции проекта появится возможность щелкнуть на кнопке Show all Files (Показать все файлы) в окне Solution Explorer и просмотреть скопированную папку \Images внутри \bin\Debug (может также потребоваться щелкнуть на кнопке Refresh (Обновить)).

Программная загрузка изображения

Инфраструктура WPF предоставляет класс по имени `BitmapImage`, определенный в пространстве имен `System.Windows.Media.Imaging`. Он позволяет загружать данные из файла изображения, местоположение которого представлено объектом `System.Uri`. Добавьте поле типа `List<BitmapImage>` для хранения всех изображений, а также поле типа `int` для хранения индекса изображения, показанного в текущий момент:

```
// Список файлов BitmapImage.
List<BitmapImage> _images=new List<BitmapImage>();
// Текущая позиция в списке.
private int _currImage=0;
```

Внутри обработчика события `Loaded` окна заполните список изображений и установите свойство `Source` элемента управления `Image` в первое изображение из списка:

```
private void MainWindow_OnLoaded(
    object sender, RoutedEventArgs e)
{
    try
    {
        string path=Environment.CurrentDirectory;
        // Загрузить эти изображения из диска при загрузке окна.
        _images.Add(new BitmapImage(new Uri(${path}\Images\Deer.jpg")));
        _images.Add(new BitmapImage(new Uri(${path}\Images\ Dogs.jpg")));
        _images.Add(new BitmapImage(new Uri(${path}\Images\Welcome.jpg")));
        // Показать первое изображение в списке.
        imageHolder.Source=_images[_currImage];
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Реализуйте обработчики для кнопок `Previous` и `Next`, чтобы обеспечить проход по изображениям. Когда пользователь добирается до конца списка, происходит переход в начало и наоборот.

```
private void btnPreviousImage_Click(
    object sender, RoutedEventArgs e)
{
    if (--_currImage < 0)
    {
        _currImage=_images.Count - 1;
    }
    imageHolder.Source=_images[_currImage];
}
```

```

private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
    if (++_currImage >= _images.Count)
    {
        _currImage=0;
    }
    imageHolder.Source=_images[_currImage];
}

```

Теперь можете запустить программу и переключаться между всеми изображениями.

Встраивание ресурсов приложения

Если файлы изображений необходимо встроить прямо в сборку .NET Core как двоичные ресурсы, тогда выберите файлы изображений в окне Solution Explorer (из папки \Images, а не \bin\Debug\Images) и установите свойство Build Action в Resource (Ресурс), а свойство Copy to Output Directory — в Do not copy (Не копировать), как показано на рис. 27.2.

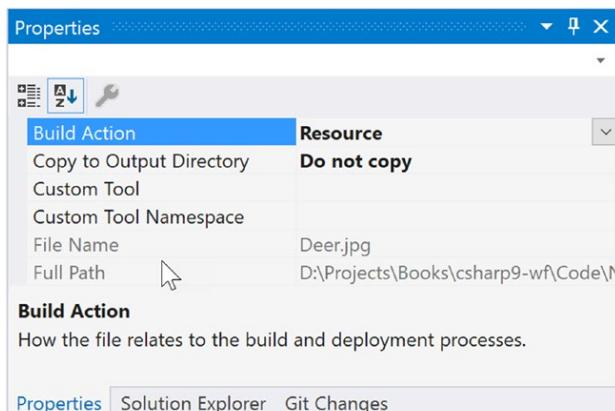


Рис. 27.2. Конфигурирование изображений как встроенных ресурсов

В меню Build (Сборка) среды Visual Studio выберите пункт Clean Solution (Очистить решение), чтобы очистить текущее содержимое папки \bin\Debug\Images, и повторно скомпилируйте проект. Обновите окно Solution Explorer и удостоверьтесь в том, что данные в каталоге \bin\Debug\Images отсутствуют. При текущих параметрах сборки графические данные больше не копируются в выходную папку, а встраиваются в саму сборку. Прием обеспечивает наличие ресурсов, но также приводит к увеличению размера скомпилированной сборки.

Вам нужно модифицировать код для загрузки изображений в список, извлекая их из скомпилированной сборки:

```

// Извлечь из сборки и затем загрузить изображения
_images.Add(new BitmapImage(new Uri(@"Images/Deer.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"Images/Dogs.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"Images/Welcome.jpg",
UriKind.Relative)));

```

В таком случае больше не придется определять путь установки и можно просто задавать ресурсы по именам, которые учитывают название исходного подкаталога. Также обратите внимание, что при создании объектов Uri указывается значение Relative перечисления UriKind. В данный момент исполняемая программа представляет собой автономную сущность, которая может быть запущена из любого местоположения на машине, т.к. все скомпилированные данные находятся внутри сборки.

Работа с объектными (логическими) ресурсами

При построении приложения WPF часто приходится определять большой объем разметки XAML для использования во многих местах окна или возможно во множестве окон либо проектов. Например, пусть создана “безупречная” кисть с линейным градиентом, определение которой в разметке занимает 10 строк. Теперь кисть необходимо применить в качестве фонового цвета для каждого элемента Button в проекте, состоящем из 8 окон, т.е. всего получается 16 элементов Button.

Худшее, что можно было бы предпринять — копировать и вставлять одну и ту же разметку XAML в каждый элемент управления Button. Очевидно, в итоге это могло бы стать настоящим кошмаром при сопровождении, т.к. всякий раз, когда нужно скорректировать внешний вид и поведение кисти, приходилось бы вносить изменения во многие места.

К счастью, *объектные ресурсы* позволяют определить фрагмент разметки XAML, назначить ему имя и сохранить в подходящем словаре для использования в будущем. Подобно двоичным ресурсам объектные ресурсы часто компилируются в сборку, где они требуются. Однако в такой ситуации нет необходимости возиться со свойством Build Action. При условии, что разметка XAML помещена в корректное местоположение, компилятор позаботится обо всем остальном.

Взаимодействие с объектными ресурсами является крупной частью процесса разработки приложений WPF. Вы увидите, что объектные ресурсы могут быть намного сложнее, чем специальная кисть. Допускается определять анимацию на основе XAML, трехмерную визуализацию, специальный стиль элемента управления, шаблон данных, шаблон элемента управления и многое другое, и упаковывать каждую сущность в многократно используемый ресурс.

Роль свойства Resources

Как уже упоминалось, для применения в приложении объектные ресурсы должны быть помещены в подходящий объект словаря. Каждый производный от FrameworkElement класс поддерживает свойство Resources, которое инкапсулирует объект ResourceDictionary, содержащий определенные объектные ресурсы. Объект ResourceDictionary может хранить элементы любого типа, потому что оперирует экземплярами System.Object и допускает манипуляций из разметки XAML или процедурного кода.

В инфраструктуре WPF все элементы управления плюс элементы Window, Page (используемые при построении навигационных приложений) и UserControl расширяют класс FrameworkElement, так что почти все виджеты предоставляют доступ к ResourceDictionary. Более того, класс Application, хотя и не расширяет FrameworkElement, но поддерживает свойство с идентичным именем Resources, которое предназначено для той же цели.

Определение ресурсов уровня окна

Чтобы приступить к исследованию роли объектных ресурсов, создайте в Visual Studio новый проект приложения WPF по имени ObjectResourcesApp и замените первоначальный элемент Grid горизонтально выровненным диспетчером компоновки StackPanel, внутри которого определите два элемента управления Button (чего вполне достаточно для пояснения роли объектных ресурсов):

```
<StackPanel Orientation="Horizontal">
    <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
    <Button Margin="25" Height="200" Width="200"
        Content="Cancel" FontSize="20"/>
</StackPanel>
```

Выберите кнопку OK и установите в свойстве Background специальный тип кисти с применением интегрированного редактора кистей (который обсуждался в главе 26). Кисть помещается внутрь области между дескрипторами <Button> и </Button>:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
    <Button.Background>
        <RadialGradientBrush>
            <GradientStop Color="#FFC44EC4" Offset="0" />
            <GradientStop Color="#FF829CEB" Offset="1" />
            <GradientStop Color="#FF793879" Offset="0.669" />
        </RadialGradientBrush>
    </Button.Background>
</Button>
```

Чтобы разрешить использовать эту кисть также и в кнопке Cancel (Отмена), область определения RadialGradientBrush должна быть расширена до словаря ресурсов родительского элемента. Например, если переместить RadialGradientBrush в StackPanel, то обе кнопки смогут применять одну и ту же кисть, т.к. они являются дочерними элементами того же самого диспетчера компоновки. Что еще лучше, кисть можно было бы упаковать в словарь ресурсов самого окна, в результате чего ее могли бы свободно использовать все элементы содержимого окна.

Когда необходимо определить ресурс, для установки свойства Resources владельца применяется синтаксис “свойство-элемент”. Кроме того, элементу ресурса назначается значение x:Key, которое будет использоваться другими частями окна для ссылки на объектный ресурс. Имейте в виду, что атрибуты x:Key и x:Name — не одно и то же! Атрибут x:Name позволяет получать доступ к объекту как к переменной-члену в файле кода, в то время как атрибут x:Key дает возможность ссылаться на элемент в словаре ресурсов.

Среда Visual Studio позволяет переместить ресурс на более высокий уровень с применением соответствующего окна Properties. Чтобы сделать это, сначала понадобится идентифицировать свойство, имеющее сложный объект, который необходимо упаковать в виде ресурса (свойство Background в рассматриваемом примере). Справа от свойства находится небольшой квадрат, щелчок на котором приводит к открытию всплывающего меню. Выберите в нем пункт Convert to New Resource (Преобразовать в новый ресурс), как продемонстрировано на рис. 27.3.

Будет запрошено имя ресурса (myBrush) и предложено указать, куда он должен быть помещен. Оставьте отмеченым переключатель This document (Этот документ), который выбирается по умолчанию (рис. 27.4).

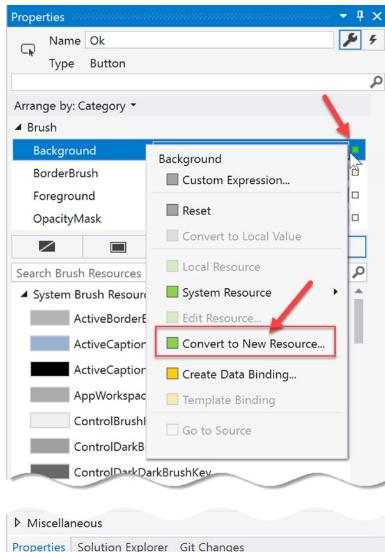


Рис. 27.3. Перемещение сложного объекта в контейнер ресурсов

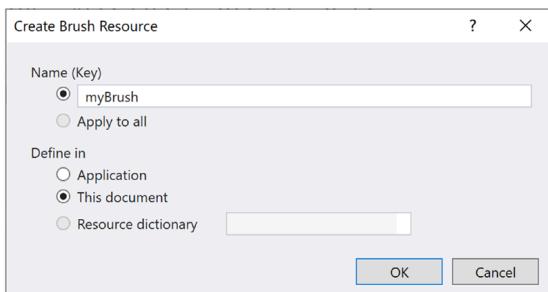


Рис. 27.4. Назначение имени объектному ресурсу

В результате определение кисти переместится внутрь дескриптора Window.Resources:

```
<Window.Resources>
<RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF829CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
</RadialGradientBrush>
</Window.Resources>
```

Свойство Background элемента управления Button обновляется для работы с новым ресурсом:

```
<Button Margin="25" Height="200" Width="200" Content="OK"
    FontSize="20" Background="{DynamicResource myBrush}" />
```

Мастер создания ресурсов определил новый ресурс как динамический (Dynamic Resource). Динамические ресурсы рассматриваются позже, а пока поменяйте тип ресурса на статический (StaticResource):

```
<Button Margin="25" Height="200" Width="200" Content="OK"
FontSize="20" Background="{StaticResource myBrush}"/>
```

Чтобы оценить преимущества, модифицируйте свойство Background кнопки Cancel (Отмена), указав в нем тот же самый ресурс StaticResource, после чего можно будет видеть повторное использование в действии:

```
<Button Margin="25" Height="200" Width="200" Content="Cancel"
FontSize="20" Background="{StaticResource myBrush}"/>
```

Расширение разметки {StaticResource}

Расширение разметки {StaticResource} применяет ресурс только один раз (при инициализации) и он остается "подключенным" к первоначальному объекту на протяжении всей жизни приложения. Некоторые свойства (вроде градиентных переходов) будут обновляться, но в случае создания нового элемента Brush, например, элемент управления не обновится. Чтобы взглянуть на такое поведение в действии, добавьте свойство Name и обработчик события Click к каждому элементу управления Button:

```
<Button Name="Ok" Margin="25" Height="200" Width="200" Content="OK"
FontSize="20" Background="{StaticResource myBrush}"
Click="Ok_OnClick"/>
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel"
FontSize="20" Background="{StaticResource myBrush}"
Click="Cancel_OnClick"/>
```

Затем поместите в обработчик события Ok_OnClick() следующий код:

```
private void Ok_OnClick(object sender, RoutedEventArgs e)
{
    // Получить кисть и внести изменения.
    var b = (RadialGradientBrush)Resources["myBrush"];
    b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
}
```

На заметку! Здесь для поиска ресурса по имени используется индексатор Resources. Тем не менее, имейте в виду, что если ресурс найти не удастся, тогда будет сгенерировано исключение времени выполнения. Можно также применять метод TryFindResource(), который не приводит к генерации исключения, а просто возвращает null, если указанный ресурс не найден.

Запустив программу и щелкнув на кнопке OK, вы заметите, что градиенты соответствующим образом изменяются. Добавьте в обработчик события Cancel_OnClick() такой код:

```
private void Cancel_OnClick(object sender, RoutedEventArgs e)
{
    // Поместить в ячейку myBrush совершенно новую кисть.
    Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

Снова запустив программу и щелкнув на кнопке Cancel, вы обнаружите, что ничего не происходит!

Расширение разметки {DynamicResource}

Для свойства также можно использовать расширение разметки DynamicResource. Чтобы выяснить разницу, измените разметку для кнопки Cancel, как показано ниже:

```
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel"
    FontSize="20" Background="{DynamicResource myBrush}"
    Click="Cancel_OnClick"/>
```

На этот раз в результате щелчка на кнопке Cancel цвет фона для кнопки Cancel изменяется, а цвет фона для кнопки OK остается прежним. Причина в том, что расширение разметки {DynamicResource} способно обнаруживать замену внутреннего объекта, указанного посредством ключа, новым объектом. Как и можно было предположить, такая возможность требует дополнительной инфраструктуры времени выполнения, так что {StaticResource} обычно следует использовать, только если не планируется заменять объектный ресурс другим объектом во время выполнения с уведомлением всех элементов, которые задействуют данный ресурс.

Ресурсы уровня приложения

Когда в словаре ресурсов окна имеются объектные ресурсы, их могут потреблять все элементы этого окна, но не другие окна приложения. Решение совместно использовать объектные ресурсы в рамках приложения предусматривает их определение на уровне приложения, а не на уровне какого-то окна. В Visual Studio отсутствуют способы автоматизации такого действия, а потому необходимо просто вырезать имеющееся определение объекта кисти из области Windows.Resource и поместить его в область Application.Resources файла App.xaml.

Теперь любое дополнительное окно или элемент управления в приложении в состоянии работать с данным объектом кисти. Ресурсы уровня приложения доступны для выбора при установке свойства Background элемента управления (рис. 27.5).

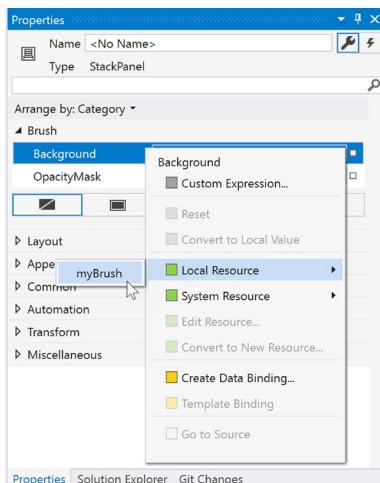


Рис. 27.5. Применение ресурсов уровня приложения

На заметку! Помещение ресурса на уровень приложения и назначение его свойству элемента управления приводит к замораживанию ресурса, что препятствует изменению значений во время выполнения. Ресурс можно клонировать и модифицировать клон.

Определение объединенных словарей ресурсов

Ресурсов уровня приложения часто оказывается вполне достаточно, но они ничем не помогут, если ресурсы необходимо разделять между проектами. В таком случае понадобится определить то, что известно как **объединенный словарь ресурсов**. Считайте его библиотекой классов для ресурсов WPF; он представляет собой всего лишь файл .xaml, содержащий коллекцию ресурсов. Единственный проект может иметь любое требуемое количество таких файлов (один для кистей, один для анимации и т.д.), каждый из которых может быть добавлен в диалоговом окне Add New Item (Добавление нового элемента), открываемом через меню Project (рис. 27.6).

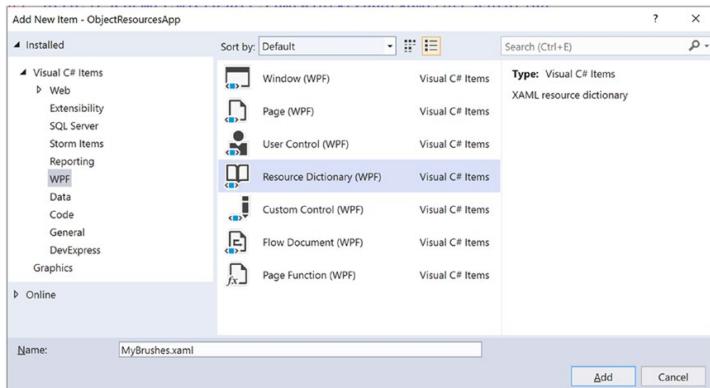


Рис. 27.6. Вставка нового объединенного словаря ресурсов

Вырежьте текущие ресурсы из области определения Application.Resources в новом файле MyBrushes.xaml и перенесите их в словарь:

```
<ResourceDictionary
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:local="clr-namespace:ObjectResourcesApp"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <RadialGradientBrush x:Key="myBrush">
        <GradientStop Color="#FFC44EC4" Offset="0" />
        <GradientStop Color="#FF827CEB" Offset="1" />
        <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
</ResourceDictionary>
```

Хотя данный словарь ресурсов является частью проекта, все словари ресурсов должны быть объединены (обычно на уровне приложения) в единый словарь ресурсов, чтобы их можно было использовать. Для этого применяется следующий формат в файле App.xaml (обратите внимание, что множество словарей ресурсов объединяются за счет добавления элементов ResourceDictionary в область ResourceDictionary.MergedDictionaries):

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Проблема такого подхода в том, что каждый файл ресурсов потребуется добавлять в каждый проект, нуждающийся в ресурсах. Более удачный подход к разделению ресурсов заключается в определении библиотеки классов .NET Core для совместного использования проектами, чем мы и займемся.

Определение сборки, включающей только ресурсы

Самый легкий способ построения сборки из одних ресурсов предусматривает создание проекта WPF User Control Library (.NET Core) (Библиотека пользовательских элементов управления WPF (.NET Core)). Создайте такой проект (по имени MyBrushesLibrary) в текущем решении, выбрав пункт меню Add⇒New Project (Добавить⇒Новый проект) в Visual Studio, и добавьте ссылку на него в проект ObjectResourcesApp.

Теперь удалите файл UserControl1.xaml из проекта. Перетащите файл MyBrushes.xaml в проект MyBrushesLibrary и удалите его из проекта ObjectResourcesApp. Наконец, откройте файл MyBrushes.xaml в проекте MyBrushesLibrary и измените пространство имен x:local на clr-namespace:MyBrushesLibrary. Вот как должно выглядеть содержимое файла MyBrushes.xaml:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:MyBrushesLibrary">
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44BC4" Offset="0" />
    <GradientStop Color="#FF829CBB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</ResourceDictionary>
```

Скомпилируйте проект WPF User Control Library. Объедините имеющиеся двоичные ресурсы со словарем ресурсов уровня приложения из проекта ObjectResourcesApp. Однако такое действие требует использования довольно забавного синтаксиса:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <!-- Синтаксис выглядит как
          /ИмяСборки;Component/ИмяФайлаXAMLвСборке.xaml -->
      <ResourceDictionary
        Source="/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Имейте в виду, что данная строка чувствительна к пробелам. Если возле символов двоеточия или косой черты будут присутствовать лишние пробелы, то возникнут

ошибки времени выполнения. Первая часть строки представляет собой дружественное имя внешней библиотеки (без файлового расширения). После двоеточия идет слово Component, а за ним имя скомпилированного двоичного ресурса, которое будет идентичным имени исходного словаря ресурсов XAML.

На этом знакомство с системой управления ресурсами WPF завершено. Описанные здесь приемы придется часто применять в большинстве разрабатываемых приложений (а то и во всех). Теперь давайте займемся исследованием встроенного API-интерфейса анимации WPF.

Службы анимации WPF

В дополнение к службам графической визуализации, которые рассматривались в главе 26, инфраструктура WPF предлагает API-интерфейс для поддержки служб анимации. Встретив термин *анимация*, многим на ум приходит вращающийся логотип компании, последовательность сменяющих друг друга изображений (для создания иллюзии движения), подпрыгивающий текст на экране или программа специфического типа вроде видеоигры или мультимедийного приложения.

Наряду с тем, что API-интерфейсы анимации WPF определенно могли бы использоваться для упомянутых выше целей, анимация может применяться всякий раз, когда приложению необходимо придать особый стиль. Например, можно было бы построить анимацию для кнопки на экране, чтобы она слегка увеличивалась, когда курсор мыши находится внутри ее границ (и возвращалась к прежним размерам, когда курсор покидает границы). Или же можно было бы предусмотреть анимацию для окна, обеспечив его закрытие с использованием определенного визуального эффекта, такого как постепенное исчезновение до полной прозрачности. Применением, более ориентированным на бизнес-приложения, может быть постепенное увеличение четкости отображения сообщений об ошибках на экране, улучшая восприятие пользовательского интерфейса. Фактически поддержка анимации WPF может применяться в приложениях любого рода (бизнес-приложениях, мультимедийных программах, видеоиграх и т.д.) всякий раз, когда нужно создать более привлекательное впечатление у пользователей.

Как и для многих других аспектов WPF, с построением анимации не связано ничего нового. Единственная особенность заключается в том, что в отличие от других API-интерфейсов, которые вы могли использовать в прошлом (включая Windows Forms), разработчики не обязаны создавать необходимую инфраструктуру вручную. В WPF не придется заранее создавать фоновые потоки или таймеры, применяемые для продвижения вперед анимационной последовательности, определять специальные типы для представления анимации, очищать и перерисовывать изображения либо реализовывать утомительные математические вычисления. Подобно другим аспектам WPF анимацию можно строить целиком в разметке XAML, целиком в коде C# либо с использованием комбинации того и другого.

На заметку! В среде Visual Studio отсутствует поддержка создания анимации посредством каких-либо графических инструментов и потому разметку XAML необходимо вводить вручную. Тем не менее, поставляемый в составе Visual Studio 2019 продукт Blend на самом деле имеет встроенный редактор анимации, который способен существенно упростить решение задач.

Роль классов анимации

Чтобы разобраться в поддержке анимации WPF, потребуется начать с рассмотрения классов анимации из пространства имен `System.Windows.Media.Animation` сборки `PresentationCore.dll`. Здесь вы найдете свыше 100 разных классов, которые содержат слово `Animation` в своих именах.

Все классы такого рода могут быть отнесены к одной из трех обширных категорий. Во-первых, любой класс, который следует соглашению об именовании вида *ТипДанных Animation* (`ByteAnimation`, `ColorAnimation`, `DoubleAnimation`, `Int32Animation` и т.д.), позволяет работать с анимацией линейной интерполяцией. Она обеспечивает плавное изменение значения во времени от начального к конечному.

Во-вторых, классы, следующие соглашению об именовании вида *ТипДанных AnimationUsingKeyFrames* (`StringAnimationUsingKeyFrames`, `DoubleAnimationUsingKeyFrames`, `PointAnimationUsingKeyFrames` и т.д.), представляют анимацию ключевыми кадрами, которая позволяет проходить в цикле по набору определенных значений за указанный период времени. Например, ключевые кадры можно применять для изменения надписи на кнопке, проходя в цикле по последовательности индивидуальных символов.

В-третьих, классы, которые следуют соглашению об именовании вида *ТипДанных AnimationUsingPath* (`DoubleAnimationUsingPath`, `PointAnimationUsingPath` и т.п.), представляют анимацию на основе пути, позволяющую перемещать объекты по определенному пути. Например, в приложении глобального позиционирования (GPS) анимацию на основе пути можно использовать для перемещения элемента по кратчайшему маршруту к месту, указанному пользователем.

Вполне очевидно, упомянутые классы не применяются для того, чтобы напрямую предоставить анимационную последовательность переменной определенного типа данных (в конце концов, как можно было бы выполнить анимацию значения 9, используя объект `Int32Animation`?).

В качестве примера возьмем свойства `Height` и `Width` типа `Label`, которые являются свойствами зависимости, упаковывающими значение `double`. Чтобы определить анимацию, которая будет увеличивать высоту метки с течением времени, можно подключить объект `DoubleAnimation` к свойству `Height` и позволить WPF позаботиться о деталях выполнения действительной анимации. Или вот другой пример: если требуется реализовать переход цвета кисти от зеленого до желтого в течение 5 секунд, то это можно сделать с применением типа `ColorAnimation`.

Следует уяснить, что классы `Animation` могут подключаться к любому свойству зависимости заданного объекта, которое имеет соответствующий тип. Как объяснялось в главе 25, свойства зависимости являются специальной формой свойств, которую требуют многие службы WPF, включая анимацию, привязку данных и стили.

По соглашению свойство зависимости определяется как статическое, доступное только для чтения поле класса, имя которого образуется добавлением слова `Property` к нормальному имени свойства. Например, для обращения к свойству зависимости для свойства `Height` класса `Button` в коде будет использоваться `Button.HeightProperty`.

Свойства To, From и By

Во всех классах `Animation` определены следующие ключевые свойства, которые управляют начальным и конечным значениями, применяемыми для выполнения анимации:

- To — представляет конечное значение анимации;
- From — представляет начальное значение анимации;
- By — представляет общую величину, на которую анимация изменяет начальное значение.

Несмотря на тот факт, что все классы поддерживают свойства To, From и By, они не получают их через виртуальные члены базового класса. Причина в том, что лежащие в основе типы, упакованные внутри указанных свойств, варьируются в широких пределах (целые числа, цвета, объекты Thickness и т.д.), и представление всех возможностей через единственный базовый класс привело бы к очень сложным кодовым конструкциям.

В связи со сказанным может возникнуть вопрос: почему не использовались обобщения .NET для определения единственного обобщенного класса анимации с одиночным параметром типа (скажем, `Animate<T>`)? Опять-таки, поскольку существует огромное количество типов данных (цвета, векторы, целые числа, строки и т.д.), применяемых для анимации свойств зависимости, решение оказалось бы не настолько ясным, как можно было бы ожидать (не говоря уже о том, что XAML обеспечивает лишь ограниченную поддержку обобщенных типов).

Роль базового класса `Timeline`

Хотя для определения виртуальных свойств To, From и By не использовался единственный базовый класс, классы `Animation` все же разделяют общий базовый класс — `System.Windows.Media.Animation.Timeline`. Данный тип предлагает набор дополнительных свойств, которые управляют темпом продвижения анимации (табл. 27.1).

Таблица 27.1. Основные свойства базового класса `Timeline`

Свойства	Описание
<code>AccelerationRatio</code> , <code>DecelerationRatio</code> , <code>SpeedRatio</code>	Эти свойства применяются для управления общим темпом анимационной последовательности
<code>AutoReverse</code>	Это свойство получает или устанавливает значение, которое указывает, должна ли временная шкала воспроизводиться в обратном направлении после завершения итерации вперед (стандартным значением является <code>false</code>)
<code>BeginTime</code>	Это свойство получает или устанавливает время запуска временной шкалы. Стандартным значением является 0, что запускает анимацию немедленно
<code>Duration</code>	Это свойство позволяет устанавливать продолжительность воспроизведения временной шкалы
<code>FileBehavior</code> , <code>RepeatBehavior</code>	Эти свойства используются для управления тем, что должно произойти после завершения временной шкалы (повторение анимации, ничего и т.д.)

Реализация анимации в коде C#

Вы построите окно, содержащее элемент `Button`, который обладает довольно странным поведением: когда на него наводится курсор мыши, он вращается вокруг

своего левого верхнего угла. Начните с создания в Visual Studio нового проекта приложения WPF по имени SpinningButtonAnimationApp. Модифицируйте начальную разметку, как показано ниже (обратите внимание на обработку события MouseEnter кнопки):

```
<Button x:Name="btnSpinner" Height="50" Width="100" Content="I Spin!"  
MouseEnter="btnSpinner_MouseEnter" Click="btnSpinner_OnClick"/>
```

В файле отделенного кода импортируйте пространство имен System.Windows.Media.Animation и добавьте в файл C# следующий код:

```
private bool _isSpinning=false;  
  
private void btnSpinner_MouseEnter(  
    object sender, MouseEventArgs e)  
{  
    if (!_isSpinning)  
    {  
        _isSpinning = true;  
        // Создать объект DoubleAnimation и зарегистрировать  
        // его с событием Completed.  
        var dblAnim = new DoubleAnimation();  
        dblAnim.Completed +=(o, s) => { _isSpinning=false; };  
        // Установить начальное и конечное значения.  
        dblAnim.From = 0;  
        dblAnim.To = 360;  
        // Создать объект RotateTransform и присвоить  
        // его свойству RenderTransform кнопки.  
        var rt = new RotateTransform();  
        btnSpinner.RenderTransform=rt;  
        // Выполнить анимацию объекта RotateTransform.  
        rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);  
    }  
}  
  
private void btnSpinner_OnClick(  
    object sender, RoutedEventArgs e)  
{  
}
```

Первая крупная задача метода btnSpinner_MouseEnter() связана с конфигурированием объекта DoubleAnimation, который будет начинать со значения 0 и заканчивать значением 360. Обратите внимание, что для этого объекта также обрабатывается событие Completed, где переключается булевская переменная уровня класса, которая применяется для того, чтобы выполняющаяся анимация не была сброшена в начало.

Затем создается объект RotateTransform, который подключается к свойству RenderTransform элемента управления Button (btnSpinner). Наконец, объект RenderTransform информируется о начале анимации его свойства Angle с использованием объекта DoubleAnimation. Реализация анимации в коде обычно осуществляется путем вызова метода BeginAnimation() и передачи ему лежащего в основе свойства зависимости, к которому необходимо применить анимацию (вспомните, что по соглашению оно определено как статическое поле класса), и связанного объекта анимации.

Добавьте в программу еще одну анимацию, которая заставит кнопку после щелчка плавно становиться невидимой. Для начала создайте обработчик события Click кнопки btnSpinner с приведенным ниже кодом:

```
private void btnSpinner_OnClick(object sender, RoutedEventArgs e)
{
    var dblAnim = new DoubleAnimation
    {
        From = 1.0,
        To = 0.0
    };
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

В коде обработчика события btnSpinner_Click() изменяется свойство Opacity, чтобы постепенно скрыть кнопку из виду. Однако в настоящий момент это затруднительно, потому что кнопка вращается слишком быстро. Как можно управлять ходом анимации? Ответ на вопрос ищите ниже.

Управление темпом анимации

По умолчанию анимация будет занимать приблизительно одну секунду для перехода между значениями, которые присвоены свойствам From и To. Следовательно, кнопка располагает одной секундой, чтобы повернуться на 360 градусов, и в то же время в течение одной секунды она постепенно скроется из виду (после щелчка на ней).

Определить другой период времени для перехода анимации можно посредством свойства Duration объекта анимации, которому присваивается объект Duration. Обычно промежуток времени устанавливается путем передачи объекта TimeSpan конструктору класса Duration. Взгляните на показанное далее изменение, в результате которого кнопке будет выделено четыре секунды на вращение:

```
private void btnSpinner_MouseEnter(
    object sender, MouseEventArgs e)
{
    if (!_isSpinning)
    {
        _isSpinning = true;

        // Создать объект DoubleAnimation и зарегистрировать
        // его с событием Completed.
        var dblAnim = new DoubleAnimation();
        dblAnim.Completed += (o, s) => { _isSpinning=false; };

        // На завершение поворота кнопке отводится четыре секунды.
        dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
        ...
    }
}
```

Благодаря такой модификации у вас должен появиться шанс щелкнуть на кнопке во время ее вращения, после чего она плавно исчезнет.

На заметку! Свойство BeginTime класса Animation также принимает объект TimeSpan.

Вспомните, что данное свойство можно устанавливать для указания времени ожидания перед запуском анимационной последовательности.

Запуск в обратном порядке и циклическое выполнение анимации

За счет установки в `true` свойства `AutoReverse` объектам `Animation` указывается о необходимости запуска анимации в обратном порядке по ее завершении. Например, если необходимо, чтобы кнопка снова стала видимой после исчезновения, можно написать следующий код:

```
private void btnSpinner_OnClick(object sender, RoutedEventArgs e)
{
    DoubleAnimation dblAnim=new DoubleAnimation
    {
        From = 1.0,
        To = 0.0
    };
    // После завершения запустить в обратном порядке.
    dblAnim.AutoReverse = true;
    btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

Если нужно, чтобы анимация повторялась несколько раз (или никогда не прекращалась), тогда можно воспользоваться свойством `RepeatBehavior`, общим для всех классов `Animation`. Передавая конструктору простое числовое значение, можно указать жестко закодированное количество повторений. С другой стороны, если передать конструктору объект `TimeSpan`, то можно задать время, в течение которого анимация должна повторяться. Наконец, чтобы выполнять анимацию бесконечно, свойство `RepeatBehavior` можно установить в `RepeatBehavior.Forever`. Взгляните на следующие способы изменения поведения повтора одного из двух объектов `DoubleAnimation`, применяемых в примере:

```
// Повторять бесконечно.
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// Повторять три раза.
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// Повторять в течение 30 секунд.
dblAnim.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

Итак, исследование приемов добавления анимации к аспектам какого-то объекта с использованием кода C# и API-интерфейса анимации WPF завершено. Теперь посмотрим, как делать то же самое с помощью разметки XAML.

Реализация анимации в разметке XAML

Реализация анимации в разметке подобна ее реализации в коде, по крайней мере, для простых анимационных последовательностей. Когда необходимо создать более сложную анимацию, которая включает изменение значений множества свойств одновременно, объем разметки может заметно увеличиться. Даже в случае применения какого-то инструмента для генерирования анимации, основанной на разметке XAML, важно знать основы представления анимации в XAML, поскольку тогда облегчается задача модификации и настройки сгенерированного инструментом содержимого.

На заметку! В подкаталоге `XamlAnimations` внутри `Chapter_27` есть несколько файлов XAML. Скопируйте их содержимое в редактор Kaxaml, чтобы просмотреть результаты.

Большой частью создание анимации подобно всему тому, что вы уже видели: по-прежнему производится конфигурирование объекта `Animation`, который затем ассоциируется со свойством объекта. Тем не менее, крупное отличие связано с тем, что разметка XAML не является дружественной к вызовам методов. В результате вместо вызова `BeginAnimation()` используется *раскадровка* как промежуточный уровень.

Давайте рассмотрим полный пример анимации, определенной в терминах XAML, и подробно ее проанализируем. Приведенное далее определение XAML будет отображать окно, содержащее единственную метку. После того как объект `Label` загрузился в память, он начинает анимационную последовательность, во время которой размер шрифта увеличивается от 12 до 100 точек за период в четыре секунды. Анимация будет повторяться столько времени, сколько объект остается загруженным в память. Разметка находится в файле `GrowLabelFont.xaml`, так что его содержимое необходимо скопировать в редактор Kaxaml, нажать клавишу `<F5>` и понаблюдать за поведением.

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="200" Width="600" WindowStartupLocation="CenterScreen"
    Title="Growing Label Font!">
    <StackPanel>
        <Label Content="Interesting...">
            <Label.Triggers>
                <EventTrigger RoutedEvent="Label.Loaded">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard TargetProperty="FontSize">
                                <DoubleAnimation From="12" To="100" Duration="0:0:4"
                                    RepeatBehavior="Forever"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Label.Triggers>
        </Label>
    </StackPanel>
</Window>

```

А теперь подробно разберем пример.

Роль раскадровок

При продвижении от самого глубоко вложенного элемента наружу первым встречается элемент `<DoubleAnimation>`, обращающийся к тем же самым свойствам, которые устанавливались в процедурном коде (`From`, `To`, `Duration` и `RepeatBehavior`):

```

<DoubleAnimation From="12" To="100" Duration="0:0:4"
    RepeatBehavior="Forever"/>

```

Как упоминалось ранее, элементы `Animation` помещаются внутрь элемента `Storyboard`, применяемого для отображения объекта анимации на заданное свойство родительского типа через свойство `TargetProperty`, которым в данном случае является `FontSize`. Элемент `Storyboard` всегда находится внутри родительского элемента по имени `BeginStoryboard`:

```
<BeginStoryboard>
  <Storyboard TargetProperty="FontSize">
    <DoubleAnimation From="12" To="100" Duration="0:0:4"
      RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>
```

Роль триггеров событий

После того как элемент `BeginStoryboard` определен, должно быть указано действие какого-то вида, которое приведет к запуску анимации. Инфраструктура WPF предлагает несколько разных способов реагирования на условия времени выполнения в разметке, один из которых называется *триггером*. С высокогуровневой точки зрения триггер можно считать способом реагирования на событие в разметке XAML без необходимости в написании процедурного кода.

Обычно когда ответ на событие реализуется в C#, пишется специальный код, который будет выполнен при поступлении события. Однако триггер — всего лишь способ получить уведомление о том, что некоторое событие произошло (загрузка элемента в память, наведение на него курсора мыши, получение им фокуса и т.д.).

Получив уведомление о появлении события, можно запускать раскадровку. В показанном ниже примере обеспечивается реагирование на факт загрузки элемента `Label` в память. Поскольку вас интересует событие `Loaded` элемента `Label`, элемент `EventTrigger` помещается в коллекцию триггеров элемента `Label`:

```
<Label Content="Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent="Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty="FontSize">
            <DoubleAnimation From="12" To="100" Duration="0:0:4"
              RepeatBehavior="Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

Рассмотрим еще один пример определения анимации в XAML, на этот раз анимации ключевыми кадрами.

Анимация с использованием дискретных ключевых кадров

В отличие от объектов анимации линейной интерполяцией, обеспечивающих только перемещение между начальной и конечной точками, объекты анимации *ключевыми кадрами* позволяют создавать коллекции специальных значений, которые должны достигаться в определенные моменты времени.

Чтобы проиллюстрировать применение типа дискретного ключевого кадра, предположим, что необходимо построить элемент управления `Button`, который выполняет анимацию своего содержимого так, что на протяжении трех секунд появляется значение `OK!` по одному символу за раз. Представленная далее разметка находится в файле `StringAnimation.xaml`. Ее можно скопировать в редактор `Kaxaml` и просмотреть результаты.

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="100" Width="300"
    WindowStartupLocation="CenterScreen" Title="Animate String Data!">
    <StackPanel>
        <Button Name="myButton" Height="40"
            FontSize="16pt" FontFamily="Verdana" Width="100">
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.Loaded">
                    <BeginStoryboard>
                        <Storyboard>
                            <StringAnimationUsingKeyFrames RepeatBehavior="Forever"
                                Storyboard.TargetProperty="Content"
                                Duration="0:0:3">
                                <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                                <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                                <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                                <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
                            </StringAnimationUsingKeyFrames>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </StackPanel>
</Window>

```

Первым делом обратите внимание, что для кнопки определяется триггер события, который обеспечивает запуск раскадровки при загрузке кнопки в память. Класс `StringAnimationUsingKeyFrames` отвечает за изменение содержимого кнопки через значение `Storyboard.TargetProperty`.

Внутри элемента `StringAnimationUsingKeyFrames` определены четыре элемента `DiscreteStringKeyFrame`, которые изменяют свойство `Content` на протяжении двух секунд (длительность, установленная объектом `StringAnimationUsingKeyFrames`, составляет в сумме три секунды, поэтому между финальным символом `!` и следующим появлением `O` будет заметна небольшая пауза).

Теперь, когда вы получили некоторое представление о том, как строятся анимации в коде C# и разметке XAML, давайте выясним роль стилей WPF, которые интенсивно задействуют графику, объектные ресурсы и анимацию.

Роль стилей WPF

При построении пользовательского интерфейса приложения WPF нередко требуется обеспечить общий вид и поведение для целого семейства элементов управления. Например, может понадобиться сделать так, чтобы все типы кнопок имели ту же самую высоту, ширину, цвет и размер шрифта для своего строкового содержимого. Хотя решить задачу можно было бы установкой идентичных значений в индивидуальных свойствах, такой подход затрудняет внесение изменений, потому что при каждом изменении придется переустанавливать один и тот же набор свойств во множестве объектов.

К счастью, инфраструктура WPF предлагает простой способ ограничения внешнего вида и поведения связанных элементов управления с использованием стилей. Выражаясь просто, стиль WPF — это объект, который поддерживает коллекцию пар “свойство-значение”. С точки зрения программирования отдельный стиль представляется с помощью класса `System.Windows.Style`. Класс `Style` имеет свойство по имени `Setters`, которое открывает доступ к строго типизированной коллекции объектов `Setter`. Именно объект `Setter` обеспечивает возможность определения пар “свойство-значение”.

В дополнение к коллекции `Setters` класс `Style` также определяет несколько других важных членов, которые позволяют встраивать триггеры, ограничивать место применения стиля и даже создавать новый стиль на основе существующего (вспомните такой прием как “наследование стилей”). Ниже перечислены наиболее важные члены класса `Style`:

- `Triggers` — открывает доступ к коллекции объектов триггеров, которая делает возможной фиксацию условий возникновения разнообразных событий в стиле;
- `BasedOn` — разрешает строить новый стиль на основе существующего;
- `TargetType` — позволяет ограничивать место применения стиля.

Определение и применение стиля

Почти в каждом случае объект `Style` упаковывается как объектный ресурс. Подобно любому объектному ресурсу его можно упаковывать на уровне окна или на уровне приложения, а также внутри выделенного словаря ресурсов (что замечательно, поскольку делает объект `Style` легко доступным во всех местах приложения). Вспомните, что цель заключается в определении объекта `Style`, который наполняет (минимум) коллекцию `Setters` набором пар “свойство-значение”.

Давайте построим стиль, который фиксирует базовые характеристики шрифта элемента управления в нашем приложении. Начните с создания в Visual Studio нового проекта приложения WPF по имени `WpfStyles`. Откройте файл `App.xaml` и определите в нем следующий именованный стиль:

```
<Application.Resources>
  <Style x:Key="BasicControlStyle">
    <Setter Property="Control.FontSize" Value="14"/>
    <Setter Property="Control.Height" Value="40"/>
    <Setter Property="Control.Cursor" Value="Hand"/>
  </Style>
</Application.Resources>
```

Обратите внимание, что объект `BasicControlStyle` добавляет во внутреннюю коллекцию три объекта `Setter`. Теперь примените получившийся стиль к нескольким элементам управления в главном окне. Из-за того, что стиль является объектным ресурсом, элементы управления, которым он необходим, по-прежнему должны использовать расширение разметки `{StackResource}` или `{DynamicResource}` для нахождения стиля. Когда они находят стиль, то устанавливают элемент ресурса в идентично именованное свойство `Style`. Замените стандартный элемент управления `Grid` следующей разметкой:

```
<StackPanel>
  <Label x:Name="lblInfo" Content="This style is boring..." 
        Style="{StaticResource BasicControlStyle}" Width="150"/>
```

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"  
       Style="{StaticResource BasicControlStyle}" Width="250"/>  
</StackPanel>
```

Если вы просмотрите элемент Window в визуальном конструкторе Visual Studio (или запустите приложение), то обнаружите, что оба элемента управления поддерживают те же самые курсор, высоту и размер шрифта.

Переопределение настроек стиля

В то время как оба элемента управления подчиняются стилю, после применения стиля к элементу управления вполне допустимо изменять некоторые из определенных настроек. Например, элемент Button теперь использует курсор Help (вместо курсора Hand, определенного в стиле):

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"  
       Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="250" />
```

Стили обрабатываются перед настройками индивидуальных свойств элемента управления, к которому применен стиль; следовательно, элементы управления могут "переопределять" настройки от случая к случаю.

Влияние атрибута `TargetType` на стили

В настоящий момент наш стиль определен так, что его может задействовать любой элемент управления (и он должен делать это явно, устанавливая свое свойство `Style`), поскольку каждое свойство уточнено посредством класса `Control`. Для программы, определяющей десятки настроек, в результате получился бы значительный объем повторяющегося кода. Один из способов несколько улучшить ситуацию предусматривает использование атрибута `TargetType`. Добавление атрибута `TargetType` к открывающему дескриптору `Style` позволяет точно указать, где стиль может быть применен (в данном примере внутри файла App.xaml):

```
<Style x:Key="BasicControlStyle" TargetType="Control">  
  <Setter Property="FontSize" Value="14"/>  
  <Setter Property="Height" Value="40"/>  
  <Setter Property="Cursor" Value="Hand"/>  
</Style>
```

На заметку! При построении стиля, использующего базовый класс, нет нужды беспокоиться о том, что значение присваивается свойству зависимости, которое не поддерживается производными типами. Если производный тип не поддерживает заданное свойство зависимости, то оно игнорируется.

Кое в чем прием помог, но все равно вы имеете стиль, который может применяться к любому элементу управления. Атрибут `TargetType` более удобен, когда необходимо определить стиль, который может быть применен только кциальному типу элементов управления. Добавьте в словарь ресурсов приложения следующий стиль:

```
<Style x:Key="BigGreenButton" TargetType="Button">  
  <Setter Property="FontSize" Value="20"/>  
  <Setter Property="Height" Value="100"/>  
  <Setter Property="Width" Value="100"/>  
  <Setter Property="Background" Value="DarkGreen"/>  
  <Setter Property="Foreground" Value="Yellow"/>  
</Style>
```



Рис. 27.7. Элементы управления с разными стилями

определения стиля при условии, что свойство `x:Key` отсутствует.

Вот еще один стиль уровня приложения, который будет автоматически применяться ко всем элементам управления `TextBox` в текущем приложении:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="30"/>
    <Setter Property="BorderThickness" Value="5"/>
    <Setter Property="BorderBrush" Value="Red"/>
    <Setter Property="FontStyle" Value="Italic"/>
</Style>
```

Теперь можно определять любое количество элементов управления `TextBox`, и все они автоматически получат установленный внешний вид. Если какому-то элементу управления `TextBox` не нужен такой стандартный внешний вид, тогда он может отказаться от него, установив свойство `Style` в `{x:Null}`. Например, элемент `txtTest` будет иметь неименованный стандартный стиль, а элемент `txtTest2` сделает все самостоятельно:

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
    BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

Создание подклассов существующих стилей

Новые стили можно также строить на основе существующего стиля посредством свойства `BasedOn`. Расширяемый стиль должен иметь подходящий атрибут `x:Key` в словаре, т.к. производный стиль будет ссылаться на него по имени, используя расширение разметки `{StaticResource}` или `{DynamicResource}`. Ниже представлен новый стиль, основанный на стиле `BigGreenButton`, который поворачивает элемент управления `Button` на 20 градусов:

```
<!-- Этот стиль основан на BigGreenButton -->
<Style x:Key="TiltButton" TargetType="Button"
    BasedOn="{StaticResource BigGreenButton}">
```

Такой стиль будет работать только с элементами управления `Button` (или подклассами `Button`). Если применить его к несовместимому элементу, тогда возникнут ошибки разметки и компиляции. Добавьте элемент управления `Button`, который использует новый стиль:

```
<Button x:Name="btnAnotherButton"
    Content="OK!" Margin="0,10,0,0"
    Style="{StaticResource BigGreenButton}"
    Width="250" Cursor="Help"/>
```

Результирующий вывод представлен на рис. 27.7.

Еще один эффект от атрибута `TargetType` заключается в том, что стиль будет применен ко всем элементам данного типа внутри области

```

<Setter Property="Foreground" Value="White"/>
<Setter Property="RenderTransform">
    <Setter.Value>
        <RotateTransform Angle="20"/>
    </Setter.Value>
</Setter>
</Style>

```

Чтобы применить новый стиль, модифицируйте разметку для кнопки следующим образом:

```

<Button x:Name="btnAnotherButton"
        Content="OK!" Margin="0,10,0,0"
        Style="{StaticResource TitlButton}"
        Width="250" Cursor="Help"/>

```

Такое действие изменяет внешний вид изображения, как показано на рис. 27.8.



Рис. 27.8. Использование производного стиля

Определение стилей с триггерами

Стили WPF могут также содержать триггеры за счет упаковки объектов Trigger в коллекцию Triggers объекта Style. Использование триггеров в стиле позволяет определять некоторые элементы Setter таким образом, что они будут применяться только в случае истинности заданного условия триггера. Например, возможно требуется увеличивать размер шрифта, когда курсор мыши находится над кнопкой. Или, скажем, нужно подсветить текстовое поле, имеющее фокус, с использованием фона указанного цвета. Триггеры полезны в ситуациях подобного рода, потому что они позволяют предпринимать специфические действия при изменении свойства, не требуя написания явной логики C# в файле отделенного кода.

Далее приведена модифицированная разметка для стиля элементов управления типа TextBox, где обеспечивается установка фона желтого цвета, когда элемент TextBox получает фокус:

```

<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="30"/>
    <Setter Property="BorderThickness" Value="5"/>
    <Setter Property="BorderBrush" Value="Red"/>
    <Setter Property="FontStyle" Value="Italic"/>
    <!-- Следующий установщик будет применен, только
         когда текстовое поле находится в фокусе -->
    <Style.Triggers>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="Background" Value="Yellow"/>
        </Trigger>
    </Style.Triggers>
</Style>

```

При тестировании этого стиля вы обнаружите, что по мере перехода с помощью клавиши `<Tab>` между элементами TextBox текущий выбранный TextBox получает фон желтого цвета (если только стиль не отключен путем присваивания `{x:Null}` свойству Style).

Триггеры свойств также весьма интеллектуальны в том смысле, что когда условие триггера *не истинно*, то свойство автоматически получает стандартное значение. Следовательно, как только TextBox теряет фокус, он также автоматически принимает стандартный цвет без какой-либо работы с вашей стороны. По контрасту с ними триггеры событий (которые исследовались при рассмотрении анимации WPF) не возвращаются автоматически в предыдущее состояние.

Определение стилей с множеством триггеров

Триггеры могут быть спроектированы так, что определенные элементы Setter будут применяться, когда истинными должны оказаться *многие условия*. Пусть необходимо устанавливать фон элемента TextBox в Yellow только в случае, если он имеет активный фокус и курсор мыши находится внутри его границ. Для этого можно воспользоваться элементом MultiTrigger и определить в нем каждое условие:

```
<!-- Стандартный стиль для всех текстовых полей -->
<Style TargetType="TextBox">
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="30"/>
    <Setter Property="BorderThickness" Value="5"/>
    <Setter Property="BorderBrush" Value="Red"/>
    <Setter Property="FontStyle" Value="Italic"/>
    <!-- Следующий установщик будет применен, только когда текстовое
        поле имеет фокус И над ним находится курсор мыши -->
    <Style.Triggers>
        <MultiTrigger>
            <MultiTrigger.Conditions>
                <Condition Property="IsFocused" Value="True"/>
                <Condition Property="IsMouseOver" Value="True" />
            </MultiTrigger.Conditions>
            <Setter Property="Background" Value="Yellow"/>
        </MultiTrigger>
    </Style.Triggers>
</Style>
```

Стили с анимацией

Стили также могут содержать в себе триггеры, которые запускают анимационную последовательность. Ниже показан последний стиль, который после применения к элементам управления Button заставит их увеличиваться и уменьшаться в размерах, когда курсор мыши находится внутри границ кнопки:

```
<!-- Стиль увеличивающейся кнопки -->
<Style x:Key="GrowingButtonStyle" TargetType="Button">
    <Setter Property="Height" Value="40"/>
    <Setter Property="Width" Value="100"/>
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Trigger.EnterActions>
                <BeginStoryboard>
                    <Storyboard TargetProperty="Height">
                        <DoubleAnimation From="40" To="200"
                            Duration="0:0:2" AutoReverse="True"/>

```

```

        </Storyboard>
    </BeginStoryboard>
</Trigger.EnterActions>
</Trigger>
</Style.Triggers>
</Style>

```

Здесь коллекция Triggers наблюдает за тем, когда свойство IsMouseOver возвратит значение true. После того как это произойдет, определяется элемент Trigger.EnterActions для выполнения простой раскладовки, которая заставляет кнопку за две секунды увеличиться до значения Height, равного 200 (и затем возвратиться к значению Height, равному 40). Чтобы отслеживать другие изменения свойств, можно также добавить область Trigger.ExitActions и определить в ней любые специальные действия, которые должны быть выполнены, когда IsMouseOver изменяется на false.

Применение стилей в коде

Вспомните, что стиль может применяться также во время выполнения. Прием удобен, когда у конечных пользователей должна быть возможность выбора внешнего вида для их пользовательского интерфейса, требуется принудительно устанавливать внешний вид и поведение на основе настроек безопасности (например, стиль DisableAllButton) или еще в какой-то ситуации.

В текущем проекте было определено порядочное количество стилей, многие из которых могут применяться к элементам управления Button. Давайте переделаем пользовательский интерфейс главного окна, чтобы позволить пользователю выбирать имена имеющихся стилей в элементе управления ListBox. На основе выбранного имени будет применен соответствующий стиль. Вот финальная разметка для элемента DockPanel:

```

<DockPanel>
    <StackPanel Orientation="Horizontal"
        DockPanel.Dock="Top" Margin="0,0,0,50">
        <Label Content="Please Pick a Style for this Button" Height="50"/>
        <ListBox x:Name="lstStyles" Height="80" Width="150"
            Background="LightBlue"
            SelectionChanged="comboStyles_Changed" />
    </StackPanel>
    <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!"/>
</DockPanel>

```

Элемент управления ListBox (по имени lstStyles) будет динамически заполняться внутри конструктора окна:

```

public MainWindow()
{
    InitializeComponent();

    // Заполнить окно со списком всеми стилями для элементов Button.
    lstStyles.Items.Add("GrowingButtonStyle");
    lstStyles.Items.Add("TiltButton");
    lstStyles.Items.Add("BigGreenButton");
    lstStyles.Items.Add("BasicControlStyle");
}

```

Последней задачей является обработка события SelectionChanged в связанном файле кода. Обратите внимание, что в следующем коде имеется возможность извлечения текущего ресурса по имени с использованием унаследованного метода TryFindResource():

```
private void comboStyles_Changed(object sender,
    SelectionChangedEventArgs e)
{
    // Получить имя стиля, выбранное в окне со списком.
    var currStyle=(Style)TryFindResource(lstStyles.SelectedValue);
    if (currStyle==null) return;

    // Установить стиль для типа кнопки.
    this.btnAdd.Style=currStyle;
}
```

После запуска приложения появляется возможность выбора одного из четырех стилей кнопок на лету. На рис. 27.9 показано готовое приложение в действии.

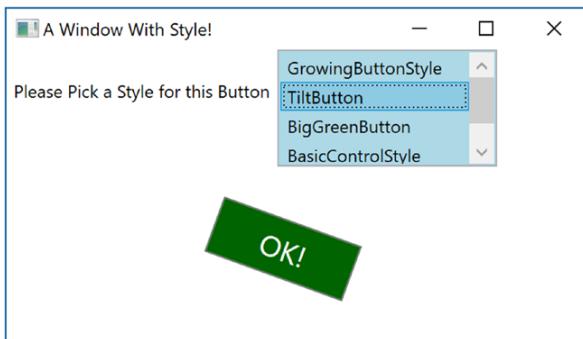


Рис. 27.9. Элементы управления с разными стилями

Логические деревья, визуальные деревья и стандартные шаблоны

Теперь, когда вы понимаете, что собой представляют стили и ресурсы, есть еще несколько тем, которые потребуется раскрыть, прежде чем приступить к изучению построения специальных элементов управления. В частности, необходимо выяснить разницу между логическим деревом, визуальным деревом и стандартным шаблоном. При вводе разметки XAML в Visual Studio или в редакторе вроде Kaxaml разметка является логическим представлением документа XAML. В случае написания кода C#, который добавляет в элемент управления StackPanel новые элементы, они вставляются в логическое дерево. По существу логическое представление отражает то, как содержимое будет позиционировано внутри разнообразных диспетчеров компоновки для главного элемента Window (или другого корневого элемента, такого как Page или NavigationWindow).

Однако за каждым логическим деревом стоит намного более сложное представление, которое называется *визуальным деревом* и внутренне применяется инфраструктурой WPF для корректной визуализации элементов на экране. Внутри любого

визуального дерева будут находиться полные детали шаблонов и стилей, используемых для визуализации каждого объекта, включая все необходимые рисунки, фигуры, визуальные объекты и объекты анимации.

Полезно уяснить разницу между логическим и визуальным деревьями, потому что при построении специального шаблона элемента управления на самом деле производится замена всего или части стандартного визуального дерева элемента управления собственным вариантом. Следовательно, если нужно, чтобы элемент управления Button визуализировался в виде звездообразной фигуры, тогда можно определить новый шаблон такого рода и подключить его к визуальному дереву Button. Логически тип остается тем же типом Button, поддерживая все ожидаемые свойства, методы и события. Но визуально он выглядит совершенно по-другому. Один лишь упомянутый факт делает WPF исключительно полезным API-интерфейсом, поскольку другие инструментальные наборы для создания кнопки звездообразной формы потребовали бы построения совершенно нового класса. В инфраструктуре WPF понадобится просто определить новую разметку.

На заметку! Элементы управления WPF часто описывают как лишенные внешности. Это относится к тому факту, что внешний вид элемента управления WPF совершенно не зависит от его поведения и допускает настройку.

Программное инспектирование логического дерева

Хотя анализ логического дерева окна во время выполнения — не слишком распространное действие при программировании с применением WPF, полезно упомянуть о том, что в пространстве имен System.Windows определен класс LogicalTreeHelper, который позволяет инспектировать структуру логического дерева во время выполнения. Для иллюстрации связи между логическими деревьями, визуальными деревьями и шаблонами элементов управления создайте новый проект приложения WPF по имени TreesAndTemplatesApp.

Замените элемент Grid приведенной ниже разметкой, которая содержит два элемента управления Button и крупный допускающий только чтение элемент TextBox с включенными линейками прокрутки. Создайте в IDE-среде обработчики событий Click для каждой кнопки. Вот результатирующая разметка XAML:

```
<DockPanel LastChildFill="True">
    <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
        <StackPanel Orientation="Horizontal">
            <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
                    Margin="4" BorderBrush="Blue" Height="40"
                    Click="btnShowLogicalTree_Click"/>
            <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
                    BorderBrush="Blue" Height="40" Click="btnShowVisualTree_Click"/>
        </StackPanel>
    </Border>
    <TextBox x:Name="txtDisplayArea" Margin="10"
            Background="AliceBlue" IsReadOnly="True"
            BorderBrush="Red" VerticalScrollBarVisibility="Auto"
            HorizontalScrollBarVisibility="Auto" />
</DockPanel>
```

Внутри файла кода C# определите переменную-член `_dataToShow` типа `string`. В обработке события `Click` объекта `btnShowLogicalTree` вызовите вспомогательную функцию, которая продолжит вызывать себя рекурсивно с целью заполнения строковой переменной логическим деревом `Window`. Для этого будет вызван статический метод `GetChildren()` объекта `LogicalTreeHelper`. Ниже показан необходимый код:

```

private string _dataToShow = string.Empty;
private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    BuildLogicalTree(0, this);
    txtDisplayArea.Text = _dataToShow;
}
void BuildLogicalTree(int depth, object obj)
{
    // Добавить имя типа к переменной-члену _dataToShow.
    _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
    // Если элемент - не DependencyObject, тогда пропустить его.
    if (!(obj is DependencyObject))
        return;
    // Выполнить рекурсивный вызов для каждого логического дочернего элемента
    foreach (var child in LogicalTreeHelper.GetChildren(
        (DependencyObject)obj))
    {
        BuildLogicalTree(depth + 5, child);
    }
}
private void btnShowVisualTree_Click(
    object sender, RoutedEventArgs e)
{
}

```

После запуска приложения и щелчка на кнопке `Logical Tree of Window` (Логическое дерево окна) в текстовой области отобразится древовидное представление, которое выглядит почти как точная копия исходной разметки XAML (рис. 27.10).

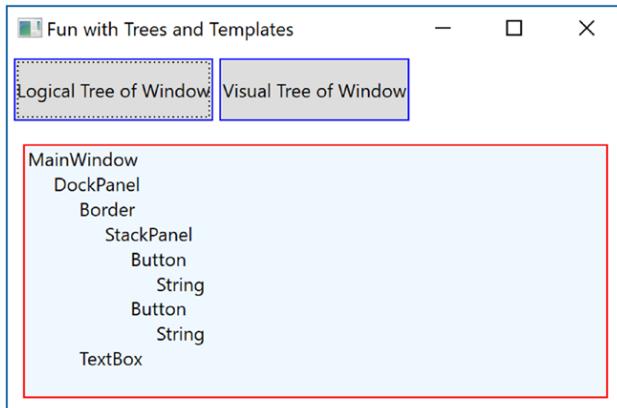


Рис. 27.10. Просмотр логического дерева во время выполнения

Программное инспектирование визуального дерева

Визуальное дерево объекта Window также можно инспектировать во время выполнения с использованием класса VisualTreeHelper из пространства имен System.Windows.Media. Далее приведена реализация обработчика события Click для второго элемента управления Button (btnShowVisualTree), которая выполняет похожую рекурсивную логику с целью построения текстового представления визуального дерева:

```
using System.Windows.Media;
private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    BuildVisualTree(0, this);
    txtDisplayArea.Text = _dataToShow;
}
void BuildVisualTree(int depth, DependencyObject obj)
{
    // Добавить имя типа к переменной-члену _dataToShow.
    _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
    //Выполнить рекурсивный вызов для каждого визуального дочернего элемента
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
    {
        BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
    }
}
```

На рис. 27.11 видно, что визуальное дерево открывает доступ к нескольким низкоуровневым агентам визуализации, таким как ContentPresenter, AdornerDecorator, TextBoxLineDrawingVisual и т.д.

Программное инспектирование стандартного шаблона элемента управления

Вспомните, что визуальное дерево применяется инфраструктурой WPF для выяснения, каким образом визуализировать элемент Window и все содержащиеся в нем элементы. Каждый элемент управления WPF хранит собственный набор команд визуализации внутри своего стандартного шаблона. С точки зрения программирования любой шаблон может быть представлен как экземпляр класса ControlTemplate. Кроме того, стандартный шаблон элемента управления можно получить через свойство Template:

```
// Получить стандартный шаблон
// элемента Button.
Button myBtn = new Button();
ControlTemplate template = myBtn.Template;
```

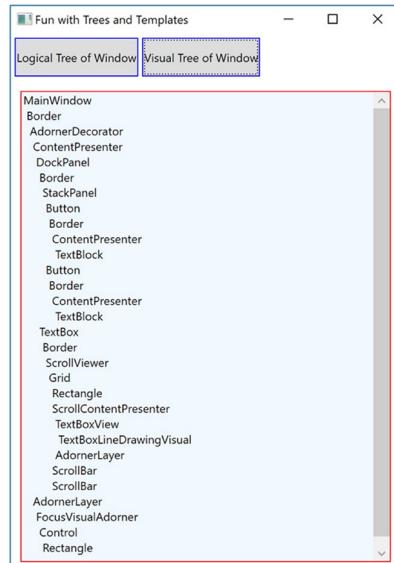


Рис. 27.11. Просмотр визуального дерева во время выполнения

Подобным же образом можно создать в коде новый объект ControlTemplate и подключить его к свойству Template элемента управления:

```
// Подключить новый шаблон для использования в кнопке.
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();
//Предположим, что этот метод добавляет весь код для звездообразного шаблона
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

Наряду с тем, что новый шаблон можно строить в коде, намного чаще это делается в разметке XAML. Тем не менее, прежде чем приступить к построению собственных шаблонов, завершите текущий пример и добавьте возможность просмотра стандартного шаблона для элемента управления WPF во время выполнения, что может оказаться полезным способом ознакомления с общей структурой шаблона. Добавьте в разметку окна новую панель StackPanel с элементами управления; она стыкована с левой стороной главной панели DockPanel (находится прямо перед элементом <TextBox>) и определена следующим образом:

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen"
       BorderThickness="4" Width="358">
    <StackPanel>
        <Label Content="Enter Full Name of WPF Control" Width="340"
              FontWeight="DemiBold" />
        <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
                 Background="BlanchedAlmond" Height="22"
                 Text="System.Windows.Controls.Button" />
        <Button x:Name="btnTemplate" Content="See Template"
               BorderBrush="Green"
               Height="40" Width="100" Margin="5"
               Click="btnTemplate_Click" HorizontalAlignment="Left" />
        <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
               Width="301" Margin="10" Background="LightGreen" >
            <StackPanel x:Name="stackTemplatePanel" />
        </Border>
    </StackPanel>
</Border>
```

Добавьте пустой обработчик события btnTemplate_Click():

```
private void btnTemplate_Click(
    object sender, RoutedEventArgs e)
{}
```

Текстовая область слева вверху позволяет вводить полностью заданное имя элемента управления WPF, расположенного в сборке PresentationFramework.dll. После того как библиотека загружена, экземпляр элемента управления динамически создается и отображается в большом квадрате слева внизу. Наконец, в текстовой области справа будет отображаться стандартный шаблон элемента управления. Добавьте в класс C# новую переменную-член типа Control:

```
private Control _ctrlToExamine = null;
```

Ниже показан оставшийся код, который требует импортирования пространств имен System.Reflection, System.Xml и System.Windows.Markup:

```

private void btnTemplate_Click(
    object sender, RoutedEventArgs e)
{
    _dataToShow = "";
    ShowTemplate();
    txtDisplayArea.Text = _dataToShow;
}

private void ShowTemplate()
{
    // Удалить элемент, который в текущий момент находится
    // в области предварительного просмотра.
    if (_ctrlToExamine != null)
        stackTemplatePanel.Children.Remove(_ctrlToExamine);
    try
    {
        // Загрузить PresentationFramework и создать экземпляр
        // указанного элемента управления. Установить его размеры для
        // отображения, а затем добавить в пустой контейнер StackPanel.
        Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0, " +
            "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
        _ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
        _ctrlToExamine.Height = 200;
        _ctrlToExamine.Width = 200;
        _ctrlToExamine.Margin = new Thickness(5);
        stackTemplatePanel.Children.Add(_ctrlToExamine);

        // Определить настройки XML для предохраниния отступов.
        var xmlSettings = new XmlWriterSettings{Indent = true};

        // Создать объект StringBuilder для хранения разметки XAML.
        var strBuilder = new StringBuilder();

        // Создать объект XmlWriter на основе имеющихся настроек.
        var xWriter = XmlWriter.Create(strBuilder, xmlSettings);

        // Сохранить разметку XAML в объекте XmlWriter на основе ControlTemplate
        XamlWriter.Save(_ctrlToExamine.Template, xWriter);

        // Отобразить разметку XAML в текстовом поле.
        _dataToShow = strBuilder.ToString();
    }
    catch (Exception ex)
    {
        _dataToShow = ex.Message;
    }
}

```

Большая часть работы связана с отображением скомпилированного ресурса BAML на строку разметки XAML. На рис. 27.12 демонстрируется финальное приложение в действии на примере вывода стандартного шаблона для элемента управления System.Windows.Controls.DatePicker. Здесь отображается календарь, который доступен по щелчку на правой части элемента управления.

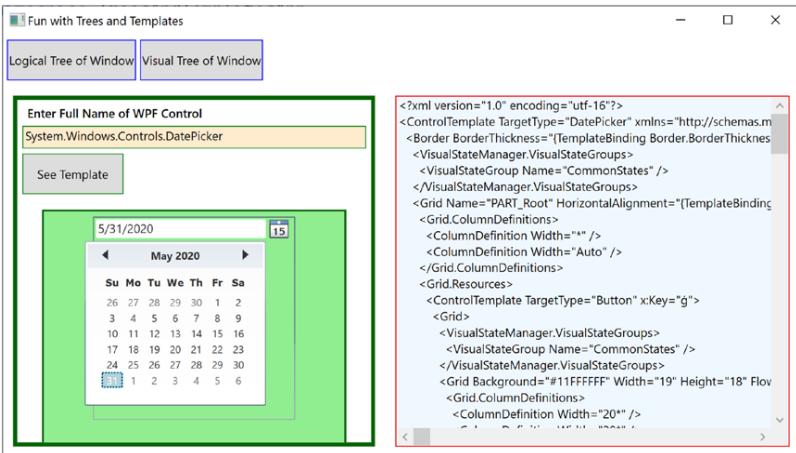


Рис. 27.12. Просмотр стандартного шаблона элемента управления во время выполнения

К настоящему моменту вы должны лучше понимать взаимосвязь между логическими деревьями, визуальными деревьями и стандартными шаблонами элементов управления. Остаток главы будет посвящен построению специальных шаблонов и пользовательских элементов управления.

Построение шаблона элемента управления с помощью инфраструктуры триггеров

Специальный шаблон для элемента управления можно создавать с помощью только кода C#. Такой подход предусматривает добавление данных к объекту `ControlTemplate` и затем присваивание его свойству `Template` элемента управления. Однако большую часть времени внешний вид и поведение `ControlTemplate` будут определяться с использованием разметки XAML и фрагментов кода (мелких или крупных) для управления поведением во время выполнения.

В оставшемся материале главы вы узнаете, как строить специальные шаблоны с применением Visual Studio. Попутно вы ознакомитесь с инфраструктурой триггеров WPF и научитесь использовать анимацию для встраивания визуальных подсказок конечным пользователям. Применение при построении сложных шаблонов только IDE-среды Visual Studio может быть связано с довольно большим объемом клавиатурного набора и трудной работы. Конечно, шаблоны производственного уровня получат преимущество от использования продукта Blend, устанавливаемого вместе с Visual Studio. Тем не менее, поскольку текущее издание книги не включает описание Blend, время засучить рукава и приступить к написанию некоторой разметки.

Для начала создайте новый проект приложения WPF по имени `ButtonTemplate`. Основной интерес в данном проекте представляют механизмы создания и применения шаблонов, так что замените элемент `Grid` следующей разметкой:

```
<StackPanel Orientation="Horizontal">
  <Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"/>
</StackPanel>
```

В обработчике события Click просто отображается окно сообщения (посредством вызова `MessageBox.Show()`) с подтверждением щелчка на элементе управления. При построении специальных шаблонов помните, что поведение элемента управления неизменно, но его внешний вид может варьироваться.

В настоящее время этот элемент Button визуализируется с использованием стандартного шаблона, который представляет собой ресурс BAML внутри заданной сборки WPF, как было проиллюстрировано в предыдущем примере. Определение собственного шаблона по существу сводится к замене стандартного визуального дерева своим вариантом. Для начала модифицируйте определение элемента Button, указав новый шаблон с применением синтаксиса “свойство-элемент”. Шаблон придаст элементу управления округлый вид.

```
<Button x:Name="myButton" Width="100" Height="100"
        Click="myButton_Click">
    <Button.Template>
        <ControlTemplate>
            <Grid x:Name="controlLayout">
                <Ellipse x:Name="buttonSurface" Fill="LightBlue"/>
                <Label x:Name="buttonCaption" VerticalAlignment="Center"
                      HorizontalAlignment="Center"
                      FontWeight="Bold" FontSize="20" Content="OK!"/>
            </Grid>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Здесь определен шаблон, который состоит из именованного элемента `Grid`, содержащего именованные элементы `Ellipse` и `Label`. Поскольку в `Grid` не определены строки и столбцы, каждый дочерний элемент укладывается поверх предыдущего элемента управления, позволяя центрировать содержимое. Если вы теперь запустите приложение, то заметите, что событие Click будет инициироваться только в ситуации, когда курсор мыши находится внутри границ элемента `Ellipse` (т.е. не на углах, окружающих эллипс). Это замечательная возможность архитектуры шаблонов WPF, т.к. нет необходимости повторно вычислять попадание курсора, проверять граничные условия или предпринимать другие низкоуровневые действия. Таким образом, если шаблон использует объект `Polygon` для отображения какой-то необычной геометрии, тогда можно иметь уверенность в том, что детали проверки попадания курсора будут соответствовать форме элемента управления, а не более крупного ограничивающего прямоугольника.

Шаблоны как ресурсы

В текущий момент ваш шаблон внедрен в специфический элемент управления `Button`, что ограничивает возможности его многократного применения. В идеале шаблон круглой кнопки следовало бы поместить в словарь ресурсов, чтобы его можно было использовать в разных проектах, или как минимум перенести в контейнер ресурсов приложения для многократного применения внутри проекта. Давайте переместим локальный ресурс `Button` на уровень приложения, вырезав определение шаблона из разметки `Button` и вставив его в дескриптор `Application.Resources` внутри файла `App.xaml`. Добавьте атрибуты `Key` и `TargetType`:

```
<Application.Resources>
    <ControlTemplate x:Key="RoundButtonTemplate"
        TargetType="{x:Type Button}">
        <Grid x:Name="controlLayout">
            <Ellipse x:Name="buttonSurface" Fill="LightBlue"/>
            <Label x:Name="buttonCaption" VerticalAlignment="Center"
                HorizontalAlignment="Center" FontWeight="Bold"
                FontSize="20" Content="OK!"/>
        </Grid>
    </ControlTemplate>
</Application.Resources>
```

Модифицируйте разметку для Button, как показано далее:

```
<Button x:Name="myButton" Width="100" Height="100"
    Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}">
</Button>
```

Из-за того, что этот ресурс доступен всему приложению, можно определять любое количество круглых кнопок, просто применяя имеющийся шаблон. В целях тестирования создайте два дополнительных элемента управления Button, которые используют данный шаблон (обрабатывать событие Click для них не нужно):

```
<StackPanel>
    <Button x:Name="myButton" Width="100" Height="100"
        Click="myButton_Click"
        Template="{StaticResource RoundButtonTemplate}"></Button>
    <Button x:Name="myButton2" Width="100" Height="100"
        Template="{StaticResource RoundButtonTemplate}"></Button>
    <Button x:Name="myButton3" Width="100" Height="100"
        Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

Встраивание визуальных подсказок с использованием триггеров

При определении специального шаблона также удаляются все визуальные подсказки стандартного шаблона. Например, стандартный шаблон кнопки содержит разметку, которая задает внешний вид элемента управления при возникновении определенных событий пользовательского интерфейса, таких как получение фокуса, щелчок кнопкой мыши, включение (или отключение) и т.д. Пользователи довольно хорошо привыкли к визуальным подсказкам подобного рода, т.к. они придают элементу управления некоторую осозаемую реакцию. Тем не менее, в шаблоне RoundButtonTemplate разметка такого типа не определена и потому внешний вид элемента управления остается идентичным независимо от действий мыши. В идеальном случае элемент должен выглядеть немного по-другому, когда на нем совершается щелчок (возможно, за счет изменения цвета или отбрасывания тени), чтобы уведомить пользователя об изменении визуального состояния.

Задачу можно решить с применением триггеров, как вы только что узнали. Для простых операций триггеры работают просто великолепно. Существуют дополнительные способы достижения цели, которые выходят за рамки настоящей книги, но больше информации доступно по адресу <https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/controls/how-to-create-apply-template>.

В качестве примера обновите шаблон RoundButtonTemplate разметкой, которая добавляет два триггера. Первый триггер будет изменять цвет фона на синий, а цвет переднего плана на желтый, когда курсор находится на поверхности элемента управления. Второй триггер уменьшит размеры элемента Grid (а также его дочерних элементов) при нажатии кнопки мыши, когда курсор расположен в пределах элемента.

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button" >
<Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" 
        FontSize="20" FontWeight="Bold"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
</Grid>
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter TargetName="buttonSurface" Property="Fill" Value="Blue"/>
        <Setter TargetName="buttonCaption" Property="Foreground" Value="Yellow"/>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
        <Setter TargetName="controlLayout"
            Property="RenderTransformOrigin" Value="0.5,0.5"/>
        <Setter TargetName="controlLayout" Property="RenderTransform">
            <Setter.Value>
                <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
            </Setter.Value>
        </Setter>
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
```

Роль расширения разметки {TemplateBinding}

Проблема с шаблоном элемента управления связана с тем, что каждая кнопка выглядит и содержит тот же самый текст. Следующее обновление разметки не оказывает никакого влияния:

```
<Button x:Name="myButton" Width="100" Height="100"
    Background="Red" Content="Howdy!" Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton2" Width="100" Height="100"
    Background="LightGreen" Content="Cancel!"
    Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton3" Width="100" Height="100"
    Background="Yellow" Content="Format"
    Template="{StaticResource RoundButtonTemplate}" />
```

Причина в том, что стандартные свойства элемента управления (такие как `Background` и `Content`) переопределяются в шаблоне. Чтобы они стали доступными, их потребуется отобразить на связанные свойства в шаблоне. Решить такие проблемы можно за счет использования расширения разметки `{TemplateBinding}` при построении шаблона. Оно позволяет захватывать настройки свойств, которые определены элементом управления, применяющим шаблон, и использовать их при установке значений в самом шаблоне.

Ниже приведена переделанная версия шаблона RoundButtonTemplate, в которой расширение разметки {TemplateBinding} применяется для отображения свойства Background элемента Button на свойство Fill элемента Ellipse; здесь также обеспечивается действительная передача значения Content элемента Button свойству Content элемента Label:

```
<Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}" />
<Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
    FontSize="20" FontWeight="Bold" HorizontalAlignment="Center"
    VerticalAlignment="Center" />
```

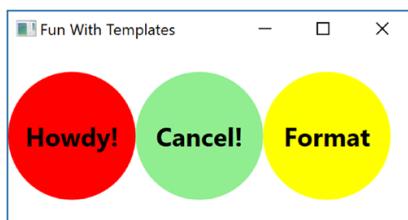


Рис. 27.13. Привязки шаблона позволяют передавать значения внутренним элементам управления

тогда можно определять элемент Button со сложным содержимым, а не только с простой строкой.

Но что, если необходимо передать сложное содержимое члену шаблона, который не имеет свойства Content? Когда в шаблоне требуется определить обобщенную область отображения содержимого, то вместо элемента управления специфического типа (Label или TextBox) можно использовать класс ContentPresenter. Хотя в рассматриваемом примере в этом нет нужды, ниже показана простая разметка, иллюстрирующая способ построения специального шаблона, который применяет класс ContentPresenter для отображения значения свойства Content элемента управления, использующего шаблон:

```
<!-- Этот шаблон кнопки отобразит то, что установлено
  в свойстве Content размещающей кнопки -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}" />
    <ContentPresenter HorizontalAlignment="Center"
        VerticalAlignment="Center" />
  </Grid>
</ControlTemplate>
```

Встраивание шаблонов в стили

В данный момент наш шаблон просто определяет базовый внешний вид и поведение элемента управления Button. Тем не менее, за процесс установки базовых свойств элемента управления (содержимого, размера шрифта, веса шрифта и т.д.) отвечает сам элемент Button:

После такого обновления появляется возможность создания кнопок с разными цветами и текстом. Результат обновления разметки XAML представлен на рис. 27.13.

Роль класса ContentPresenter

При проектировании шаблона для отображения текстового значения элемента управления использовался элемент Label. Подобно Button он поддерживает свойство Content. Следовательно, если применяется расширение разметки {TemplateBinding},

```
<!-- Сейчас базовые значения свойств должен устанавливать
    сам элемент Button, а не шаблон -->
<Button x:Name="myButton" Foreground="Black"
        FontSize="20" FontWeight="Bold"
        Template="{StaticResource RoundButtonTemplate}"
        Click="myButton_Click"/>
```

При желании значения базовых свойств можно устанавливать в шаблоне. В сущности, таким способом фактически создаются стандартный внешний вид и поведение. Как вам уже должно быть понятно, это работа стилей WPF. Когда строится стиль (для учета настроек базовых свойств), можно определить шаблон внутри стиля! Ниже показан измененный ресурс приложения внутри файла App.xaml, которому назначен ключ RoundButtonStyle:

```
<!-- Стиль, содержащий шаблон -->
<Style x:Key="RoundButtonStyle" TargetType="Button">
    <Setter Property="Foreground" Value="Black"/>
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="Width" Value="100"/>
    <Setter Property="Height" Value="100"/>
    <!-- Далее следует сам шаблон -->
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <!-- Шаблон элемента управления из предыдущего примера -->
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

После такого обновления кнопочные элементы управления можно создавать с установкой свойства Style следующим образом:

```
<Button x:Name="myButton" Background="Red" Content="Howdy!"
        Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>
```

Несмотря на то что внешний вид и поведение кнопки остаются такими же, премещество внедрения шаблонов внутрь стилей связано с тем, что появляется возможность предоставить готовый набор значений для общих свойств. На этом обзор применения Visual Studio и инфраструктуры триггеров при построении специальных шаблонов для элемента управления завершен. Хотя об инфраструктуре WPF можно еще много чего сказать, теперь у вас имеется хороший фундамент для дальнейшего самостоятельного изучения.

Резюме

Первой в главе рассматривалась система управления ресурсами WPF. Мы начали с исследования работы с двоичными ресурсами и роли объектных ресурсов. Вы узнали, что объектные ресурсы представляют собой именованные фрагменты разметки XAML, которые могут быть сохранены в разнообразных местах с целью многократного использования содержимого.

Затем был описан API-интерфейс анимации WPF. В приведенных примерах анимация создавалась с помощью кода C#, а также посредством разметки XAML. Для управления выполнением анимации, определенной в разметке, применяются элементы Storyboard и триггеры. Далее был продемонстрирован механизм стилей WPF, который интенсивно использует графику, объектные ресурсы и анимацию.

После этого вы прояснили отношение между логическим и визуальным деревьями. В своей основе логическое дерево является однозначным соотвествием разметке, которая создана для описания корневого элемента WPF. Позади логического дерева находится гораздо более глубокое визуальное дерево, содержащее детальные инструкции визуализации.

Кроме того, вы изучили роль стандартного шаблона. Не забывайте, что при построении специальных шаблонов вы по существу заменяете все визуальное дерево элемента управления (или часть дерева) собственной реализацией.

ГЛАВА 28

Уведомления WPF, проверка достоверности, команды и MVVM

В настоящей главе исследование программной модели WPF завершается рассмотрением возможностей, которые поддерживаются паттерном “модель–представление–модель представления” (Model View ViewModel — MVVM). Вы также узнаете о системе уведомлений WPF и ее реализации паттерна “Наблюдатель” (Observer) через наблюдаемые модели и коллекции. Обеспечение автоматического отображения пользовательским интерфейсом текущего состояния данных значительно улучшает его восприятие конечными пользователями и сокращает объем ручного кодирования, требуемого для получения того же результата с помощью более старых технологий (вроде Windows Forms).

Во время разработки на основе паттерна “Наблюдатель” вы ознакомитесь с механизмами добавления проверки достоверности в свои приложения. Проверка достоверности — жизненно важная часть любого приложения, которая позволяет не только сообщать пользователю о том, что что-то пошло не так, но и указывать, в чем именно заключается проблема. Вы научитесь встраивать проверку достоверности в разметку представления для информирования пользователя о возникающих ошибках.

Затем вы более глубоко погрузитесь в систему команд WPF и создадите специальные команды для инкапсуляции программной логики почти так, как поступали в главе 25 со встроенными командами. С созданием специальных команд связано несколько преимуществ, включая (помимо прочего) возможность многократного использования кода, инкапсуляцию логики и разделение обязанностей.

Наконец, вы задействуете все это в примере приложения MVVM.

Введение в паттерн MVVM

Прежде чем приступить к детальному исследованию уведомлений, проверки достоверности и команд в WPF, было бы неплохо пролить свет на конечную цель настоящей главы, которой является паттерн “модель–представление–модель представления” (MVVM). Будучи производным от паттерна проектирования “Модель представления” (Presentation Model) Мартина Фаулера, паттерн MVVM задействует обсуждаемые в главе возможности, специфичные для XAML, чтобы сделать процесс разработки приложений WPF более быстрым и ясным. Само название паттерна отражает его основные компоненты: модель (Model), представление (View) и модель представления (ViewModel).

Модель

Модель — это объектное представление имеющихся данных. В паттерне MVVM модели концептуально совпадают с моделями внутри нашего уровня доступа к данным (Data Access Layer — DAL). Иногда они являются теми же физическими классами, но поступать так вовсе не обязательно. По мере чтения главы вы узнаете, каким образом решать, применять ли модели DAL или же создавать новые модели.

Модели обычно используют в своих интересахстроенную (либо специальную) проверку достоверности через аннотации данных и интерфейс `INotifyDataErrorInfo` и сконфигурированы как наблюдаемые классы для связывания с системой уведомлений WPF. Все упомянутые темы рассматриваются позже в главе.

Представление

Представление — это пользовательский интерфейс приложения, который спроектирован так, чтобы быть чрезвычайно легковесным. Вспомните о стенде меню в ресторане для автомобилистов. На стенде отображаются позиции меню и цены, а также имеется механизм взаимодействия клиента с внутренними системами. Однако в стенд не внедрены какие-либо интеллектуальные возможности, разве что он может быть снабжен специальной логикой пользовательского интерфейса, такой как включение освещения в темное время суток.

Представления MVVM должны разрабатываться с учетом аналогичных целей. Любые интеллектуальные возможности необходимо встраивать в какие-то другие места приложения. Иметь прямое отношение к манипулированию пользовательским интерфейсом может только код в файле отделенного кода (например, в `MainWindow.xaml.cs`). Он не должен быть основан на бизнес-правилах или на чем-то еще, что нуждается в предохранении для будущего применения. Хотя это не является главной целью MVVM, хорошо разработанные приложения MVVM обычно имеют совсем небольшой объем отделенного кода.

Модель представления

В WPF и других технологиях XAML модель представления служит двум целям.

- Модель представления предлагает единственное местоположение для всех данных, необходимых представлению. Это вовсе не означает, что модель представления отвечает за получение действительных данных; замен она является просто транспортным механизмом для перемещения данных из хранилища в представление. Обычно между представлениями и моделями представлений имеется отношение “один к одному”, но существуют архитектурные отличия, которые в каждом конкретном случае могут варьироваться.
- Вторая цель модели представления касается ее действия в качестве контроллера для представления. Почти как стенд меню модель представления принимает указание от пользователя и передает их соответствующему коду для выполнения подходящих действий. Довольно часто такой код имеет форму специальных команд.

Анемичные модели или анемичные модели представлений

На заре развития WPF, когда разработчики все еще были в поиске лучшей реализации паттерна MVVM, велись бурные (а временами и жаркие) дискуссии о том, где реализовывать элементы, подобные проверке достоверности и паттерну “Наблюдатель”.

Один лагерь (сторонников анемичной (иногда называемой бескровной) модели) аргументировал, что все элементы должны находиться в моделях представлений, поскольку добавление таких возможностей к модели нарушает принцип разделения обязанностей. Другой лагерь (сторонников анемичной модели представления) утверждал, что все элементы должны находиться в моделях, т.к. тогда сокращается дублирование кода.

Конечно, фактический ответ зависит от обстоятельств. Реализация классами моделей интерфейсов `INotifyPropertyChanged`, `IDataErrorInfo` и `INotifyDataErrorInfo` гарантирует, что соответствующий код близок к своей цели (как вы увидите далее в главе) и реализован только однократно для каждой модели. Другими словами, есть ситуации, когда сами классы моделей представлений необходимо разрабатывать как наблюдаемые. По большому счету вы должны самостоятельно выяснить, что имеет больший смысл для приложения, не приводя к чрезмерному усложнению кода и не принося в жертву преимущества MVVM.

На заметку! Для WPF доступны многочисленные инфраструктуры MVVM, такие как MVVMLite, Caliburn.Micro и Prism (хотя Prism — нечто намного большее, чем просто инфраструктура MVVM). В настоящей главе обсуждается паттерн MVVM и функциональные средства WPF, которые поддерживают его реализацию. Исследование других инфраструктур и выбор среди них наиболее подходящей для нужд приложения остается за вами как разработчиком.

Система уведомлений привязки WPF

Значительным недостатком системы привязки Windows Forms является отсутствие уведомлений. Если находящиеся внутри представления данные модифицируются в коде, то пользовательский интерфейс также должен обновляться программно, чтобы оставаться в синхронном состоянии с ними. Итогом будет большое количество вызовов метода `Refresh()` на элементах управления, обычно превышающее абсолютное необходимое для обеспечения безопасности. Наряду с тем, что наличие слишком многих обращений к `Refresh()` обычно не приводит к серьезной проблеме с производительностью, недостаточное их число может отрицательно повлиять на пользовательский интерфейс.

Система привязки,строенная в приложения на основе XAML, устраняет указанную проблему за счет того, что позволяет привязывать объекты данных и коллекции к системе уведомлений, разрабатывая их как наблюдаемые. Всякий раз, когда изменяется значение свойства в наблюдаемой модели либо происходит изменение в наблюдаемой коллекции (например, добавление, удаление или переупорядочение элементов), инициируется событие (`NotifyPropertyChanged` либо `NotifyCollectionChanged`). Инфраструктура привязки автоматически прослушивает такие события и в случае их появления обновляет привязанные элементы управления. Более того, разработчики имеют контроль над тем, для каких свойств выдаются уведомления. Выглядит безупречно, не так ли? На самом деле все не настолько безупречно. Настройка наблюдаемых моделей вручную требует написания довольно большого объема кода. К счастью, как вы вскоре увидите, существует инфраструктура с открытым кодом, которая значительно упрощает работу.

Наблюдаемые модели и коллекции

В этом разделе вы построите приложение, в котором используются наблюдаемые модели и коллекции. Для начала создайте новый проект приложения WPF по имени

WpfNotifications. В приложении будет применяться форма “главная–подробности”, которая позволит пользователю выбирать объект автомобиля в элементе управления ComboBox и просматривать детальную информацию о нем в расположенных ниже элементах управления TextBox. Поместите в файл MainWindow.xaml следующую разметку:

```

<Grid IsSharedSizeScope="True" Margin="5,0,5,5">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid Grid.Row="0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label Grid.Column="0" Content="Vehicle"/>
        <ComboBox Name="cboCars" Grid.Column="1"
                  DisplayMemberPath="PetName" />
    </Grid>
    <Grid Grid.Row="1" Name="DetailsGrid">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.Row="0" Content="Id"/>
        <TextBox Grid.Column="1" Grid.Row="0" />
        <Label Grid.Column="0" Grid.Row="1" Content="Make"/>
        <TextBox Grid.Column="1" Grid.Row="1" />
        <Label Grid.Column="0" Grid.Row="2" Content="Color"/>
        <TextBox Grid.Column="1" Grid.Row="2" />
        <Label Grid.Column="0" Grid.Row="3" Content="Pet Name"/>
        <TextBox Grid.Column="1" Grid.Row="3" />
        <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="4"
                    HorizontalAlignment="Right"
                    Orientation="Horizontal" Margin="0,5,0,5">
            <Button x:Name="btnAddCar" Content="Add Car"
                   Margin="5,0,5,0" Padding="4, 2" />
            <Button x:Name="btnChangeColor" Content="Change Color"
                   Margin="5,0,5,0" Padding="4, 2"/>
        </StackPanel>
    </Grid>
</Grid>
```

Окно должно напоминать показанное на рис. 28.1.

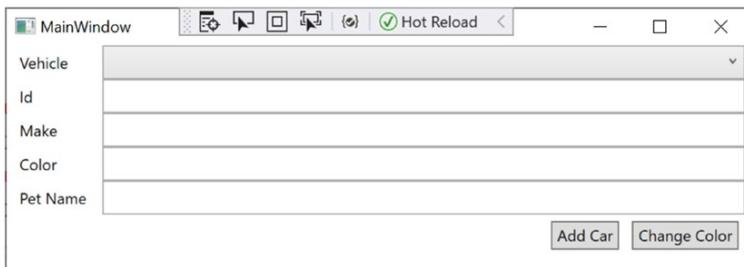


Рис. 28.1. Окно “главная–подробности”, отображающее информацию об автомобиле

Свойство `IsSharedSizeScope` элемента управления `Grid` заставляет дочерние сетки разделять размеры. Элемент `ColumnDefinitions`, помеченный как `SharedSizeGroup`, автоматически получит ту же самую ширину без каких-либо потребностей в программировании. В рассматриваемом примере, если размер метки `Pet Name` (Дружественное имя) изменяется из-за более длинного значения, тогда соответствующим образом корректируется и размер колонки `Vehicle` (Автомобиль), который находится в другом элементе управления `Grid`, сохраняя аккуратный внешний вид окна.

Щелкните правой кнопкой мыши на имени проекта в окне `Solution Explorer`, выберите в контекстном меню пункт `Add⇒New Folder` (Добавить⇒Новая папка) и назначьте новой папке имя `Models`. Создайте в новой папке файл класса `Car.cs`. Первоначально код класса выглядит так:

```
public class Car
{
    public int Id { get; set; }
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }
}
```

Добавление привязок и данных

Следующий шаг заключается в создании операторов привязки для элементов управления. Вспомните, что конструкции привязки данных врачаются вокруг контекста данных, который может быть установлен в самом элементе управления или в родительском элементе управления. Здесь контекст будет установлен в элементе `DetailsGrid`, так что каждый содержащийся внутри него элемент управления унаследует результирующий контекст данных.

Установите свойство `DataContext` в свойство `SelectedItem` элемента `ComboBox`. Модифицируйте определение элемента `Grid`, содержащего элементы управления с информацией об автомобиле, следующим образом:

```
<Grid Grid.Row="1" Name="DetailsGrid"
      DataContext="{Binding ElementName=cboCars, Path=SelectedItem}">
```

Текстовые поля в элементе `DetailsGrid` будут отображать индивидуальные характеристики выбранного автомобиля. Добавьте подходящие атрибуты `Text` и привязки к элементам управления `TextBox`:

```
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=Id}" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Make}" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=Color}" />
<TextBox Grid.Column="1" Grid.Row="3" Text="{Binding Path=PetName}" />
```

Наконец, поместите нужные данные в элемент управления ComboBox. В файле MainWindow.xaml.cs создайте новый список записей Car и присвойте его свойству ItemsSource элемента ComboBox. Кроме того, добавьте оператор using для пространства имен WpfNotifications.Models.

```
using WpfNotifications.Models;
// Для краткости код не показан.
public partial class MainWindow : Window
{
    readonly IList<Car> _cars = new List<Car>();
    public MainWindow()
    {
        InitializeComponent();
        _cars.Add(new Car { Id = 1, Color = "Blue", Make = "Chevy",
                           PetName = "Kit" });
        _cars.Add(new Car { Id = 2, Color = "Red", Make = "Ford",
                           PetName = "Red Rider" });
        cboCars.ItemsSource = _cars;
    }
}
```

Запустите приложение. Вы увидите, что в поле со списком Vehicle для выбора доступны два варианта автомобилей. Выбор одного из них приводит к автоматическому заполнению текстовых полей сведениями об автомобиле. Измените цвет одного из автомобилей, выберите другой автомобиль и затем возвратитесь к автомобилю, запись о котором редактировалась. Вы обнаружите, что новый цвет по-прежнему связан с автомобилем. Здесь нет ничего примечательного, просто демонстрируется мощь привязки данных XAML.

Изменение данных об автомобиле в коде

Несмотря на то что предыдущий пример работает ожидаемым образом, когда данные изменяются программно, пользовательский интерфейс не отразит изменения до тех пор, пока в приложении не будет предусмотрен код для обновления данных. Чтобы проиллюстрировать сказанное, добавьте обработчик события Click для кнопки btnChangeColor:

```
<Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
        Padding="4, 2" Click="BtnChangeColor_OnClick"/>
```

Внутри обработчика события BtnChangeColor_OnClick() с помощью свойства SelectedItem элемента управления ComboBox отыщите выбранную запись в списке автомобилей и измените ее цвет на Pink:

```
private void BtnChangeColor_OnClick(object sender, RoutedEventArgs e)
{
    _cars.First(x => x.Id == ((Car)cboCars.SelectedItem)?.Id).Color = "Pink";
}
```

Запустите приложение, выберите автомобиль и щелкните на кнопке Change Color (Изменить цвет). Никаких видимых изменений не произойдет. Выберите другой автомобиль и затем снова первоначальный. Теперь вы заметите обновленное значение. Для пользователя такое поведение не особенно подходит.

Добавьте обработчик события Click для кнопки btnAddCar:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0"
        Padding="4, 2" Click="BtnAddCar_OnClick" />
```

В обработчике события BtnAddCar_OnClick() добавьте новую запись в список Car:

```
private void BtnAddCar_Click(object sender, RoutedEventArgs e)
{
    var maxCount = _cars?.Max(x => x.Id) ?? 0;
    _cars?.Add(new Car
    {
        Id=++maxCount, Color="Yellow", Make="VW", PetName="Birdie"
    });
}
```

Запустите приложение, щелкните на кнопке Add Car (Добавить автомобиль) и просмотрите содержимое элемента управления ComboBox. Хотя известно, что в списке имеется три автомобиля, в элементе ComboBox отображаются только два! Чтобы устранить обе проблемы, вы превратите класс Car в наблюдаемую модель и будете использовать наблюдаемую коллекцию для хранения всех экземпляров Car.

Наблюдаемые модели

Проблема с тем, что изменение значения свойства модели не отображается в пользовательском интерфейсе, решается за счет реализации классом модели Car интерфейса INotifyPropertyChanged. Интерфейс INotifyPropertyChanged содержит единственное событие PropertyChangedEvent. Механизм привязки XAML прослушивает это событие для каждого привязанного свойства в классах, реализующих интерфейс INotifyPropertyChanged. Вот как определен интерфейс INotifyPropertyChanged:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Добавьте в файл Car.cs следующие операторы using:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
```

Затем обеспечьте реализацию классом Car интерфейса INotifyPropertyChanged:

```
public class Car : INotifyPropertyChanged
{
    // Для краткости код не показан.
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Событие PropertyChanged принимает объектную ссылку и новый экземпляр класса PropertyChangedEventArgs:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Model"));
```

Первый параметр представляет собой объект, который инициирует событие. Конструктор класса `PropertyChangedEventArgs` принимает строку, указывающую свойство, которое было изменено и нуждается в обновлении. Когда событие инициировано, механизм привязки ищет элементы управления, привязанные к именованному свойству данного объекта. В случае передачи конструктору `PropertyChangedEventArgs` значения `String.Empty` обновляются все привязанные свойства объекта.

Вы сами управляете тем, какие свойства вовлечены в процесс автоматического обновления. Автоматически обновляться будут только те свойства, которые генерируют событие `PropertyChanged` внутри блока `set`. Обычно в перечень входят все свойства классов моделей, но в зависимости от требований приложения некоторые свойства можно опускать. Вместо инициирования события `PropertyChanged` непосредственно в блоке `set` для каждого задействованного свойства распространенный подход предусматривает написание вспомогательного метода (как правило, называемого `OnPropertyChanged()`), который генерирует событие от имени свойств обычно в базовом классе для моделей. Добавьте в класс `Car` следующий метод:

```
protected void OnPropertyChanged([CallerMemberName]
    string propertyName = "")
{
    PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
}
```

Модифицируйте каждое автоматическое свойство класса `Car`, чтобы оно имело полноценные блоки `get` и `set`, а также поддерживающее поле. В случае если значение изменилось, вызовите вспомогательный метод `OnPropertyChanged()`. Вот обновленное свойство `Id`:

```
private int _id;
public int Id
{
    get => _id;
    set
    {
        if (value == _id) return;
        _id = value;
        OnPropertyChanged();
    }
}
```

Проделайте аналогичную работу со всеми остальными свойствами в классе и снова запустите приложение. Выберите автомобиль и щелкните на кнопке `Change Color`. Изменение немедленно отобразится в пользовательском интерфейсе. Первая проблема решена!

Использование операции nameof

В версии C# 6 появилась операция `nameof`, которая возвращает строковое имя переданного ей элемента. Ее можно применять в вызовах метода `OnPropertyChanged()` внутри блоков `set`, например:

```
private string _color;
public string Color
{
```

```

get { return _color; }
set
{
    if (value == _color) return;
    _color = value;
    OnPropertyChanged(nameof(Color));
}
}

```

Обратите внимание на то, что в случае использования операции nameof удалять атрибут [CallerMemberName] из метода OnPropertyChanged() не обязательно (хотя он становится излишним). В конце концов, выбор между применением операции nameof или атрибута CallerMemberName зависит от личных предпочтений.

Наблюдаемые коллекции

Следующей проблемой, которую необходимо решить, является обновление пользовательского интерфейса при изменении содержимого коллекции, что достигается путем реализации интерфейса INotifyCollectionChanged. Подобно INotifyPropertyChanged данный интерфейс открывает доступ к единственному событию CollectionChanged. В отличие от INotifyPropertyChanged реализация интерфейса INotifyCollectionChanged вручную предполагает больший объем действий, чем просто вызов метода в блоке set свойства. Понадобится создать реализацию полного списка объектов и генерировать событие CollectionChanged каждый раз, когда он изменяется.

Использование класса *ObservableCollection<T>*

К счастью, существует намного более легкий способ, чем создание собственных классов коллекций. Класс ObservableCollection<T> реализует интерфейсы INotifyCollectionChanged, INotifyPropertyChanged и Collection<T> и входит в состав .NET Core. Никакой дополнительной работы делать не придется. Чтобы продемонстрировать его применение, добавьте оператор using для пространства имен System.Collections.ObjectModel и модифицируйте закрытое поле _cars следующим образом:

```

private readonly IList<Car> _cars =
    new ObservableCollection<Car>();

```

Снова запустите приложение и щелкните на кнопке Add Car. Новые записи будут должным образом появляться.

Реализация флага изменения

Еще одним преимуществом наблюдаемых моделей является способность отслеживать изменения состояния. Отслеживать флаги изменения (т.е. когда изменяется одно и более значений объекта) в WPF довольно легко. Добавьте в класс Car свойство типа bool по имени IsChanged. Внутри его блока set вызовите метод OnPropertyChanged(), как поступали с другими свойствами класса Car.

```

private bool _isChanged;
public bool IsChanged {
    get => _isChanged;
    set
    {

```

```

    if (value == _isChanged) return;
    _isChanged = value;
    OnPropertyChanged();
}
}

```

Свойство `IsChanged` необходимо устанавливать в `true` внутри метода `OnPropertyChanged()`. Важно не устанавливать свойство `IsChanged` в `true` в случае изменения его самого, иначе генерируется исключение переполнения стека! Модифицируйте метод `OnPropertyChanged()` следующим образом (здесь используется описанная ранее операция `nameof`):

```

protected virtual void OnPropertyChanged([CallerMemberName]
    string propertyName = "")
{
    if (propertyName != nameof(IsChanged))
    {
        IsChanged = true;
    }
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}

```

Откройте файл `MainWindow.xaml` и добавьте в `DetailsGrid` дополнительный элемент `RowDefinition`. Поместите в конец элемента `Grid` показанную ниже разметку, которая содержит элементы управления `Label` и `CheckBox`, привязанные к свойству `IsChanged`:

```

<Label Grid.Column="0" Grid.Row="5" Content="Is Changed"/>
<CheckBox Grid.Column="1" Grid.Row="5" VerticalAlignment="Center"
    Margin="10,0,0,0" IsEnabled="False" IsChecked="{Binding Path=IsChanged}" />

```

Если вы запустите приложение прямо сейчас, то увидите, что каждая отдельная запись отображается как измененная, хотя пока ничего не изменилось! Дело в том, что во время создания объекта устанавливаются значения свойств, а установка любых значений приводит к вызову метода `OnPropertyChanged()`, который и устанавливает свойство `IsChanged` объекта. Чтобы устранить проблему, установите свойство `IsChanged` в `false` последним в коде инициализации объекта. Откройте файл `MainWindow.xaml.cs` и модифицируйте код создания списка:

```

_cars.Add(
    new Car {Id = 1, Color = "Blue", Make = "Chevy",
        PetName = "Kit", IsChanged = false});
_cars.Add(
    new Car {Id = 2, Color = "Red", Make = "Ford",
        PetName = "Red Rider", IsChanged = false});

```

Снова запустите приложение, выберите автомобиль и щелкните на кнопке `Change Color`. Флажок `Is Changed` (`Изменено`) становится отмеченным наряду с изменением цвета.

Обновление источника через взаимодействие с пользовательским интерфейсом

Во время выполнения приложения можно заметить, что при вводе в текстовых полях флажок `Is Changed` не становится отмеченным до тех пор, пока фокус не покинет элемент управления, где производился ввод. Причина кроется в свойстве `UpdateSourceTrigger` привязок элементов `TextBox`.

Свойство `UpdateSourceTrigger` определяет, какое событие (изменение значения, переход фокуса и т.д.) является основанием для обновления пользовательским интерфейсом лежащих в основе данных. Перечисление `UpdateSourceTrigger` принимает значения, описанные в табл. 28.1.

Таблица 28.1. Значения перечисления `UpdateSourceTrigger`

Значение	Описание
Default	Устанавливает стандартное значение, принятое для элемента управления (например, <code>LostFocus</code> для <code>TextBox</code>)
Explicit	Обновляет объект источника только при вызове метода <code>UpdateSource()</code>
LostFocus	Обновляет, когда элемент управления теряет фокус. Является стандартным для <code>TextBox</code>
PropertyChanged	Обновляет при изменении свойства. Является стандартным для <code>CheckBox</code>

Стандартным событием обновления для элементов управления `TextBox` является `LostFocus`. Измените его на `PropertyChanged`, модифицировав привязку для элемента `TextBox`, который отвечает за ввод цвета:

```
<TextBox Grid.Column="1" Grid.Row="2"
         Text="{Binding Path=Color, UpdateSourceTrigger=PropertyChanged}" />
```

Если вы запустите приложение и начнете ввод в текстовом поле `Color` (Цвет), то флажок `Is Changed` немедленно отметится. Может возникнуть вопрос о том, почему для элементов управления `TextBox` в качестве стандартного выбрано значение `LostFocus`. Дело в том, что проверка достоверности (рассматриваемая вскоре) для модели запускается в сочетании с `UpdateSourceTrigger`. В случае `TextBox` это может потенциально вызывать ошибки, которые будут постоянно возникать до тех пор, пока пользователь не введет корректное значение. Например, если правила проверки достоверности не разрешают вводить в элементе `TextBox` менее пяти символов, тогда сообщение об ошибке будет отображаться при каждом нажатии клавиши, пока пользователь не введет пять или более символов. В таких случаях с обновлением источника лучше подождать до момента, когда пользователь переместит фокус из элемента `TextBox` (завершив изменение текста).

Итоговые сведения об уведомлениях и наблюдаемых моделях

Применение интерфейсов `IPropertyChanged` в моделях и классов `ObservableCollection` для списков улучшает пользовательский интерфейс приложения за счет поддержания его в синхронизированном состоянии с данными. В то время как ни один из интерфейсов не является сложным, они требуют обновлений кода. К счастью, в инфраструктуре предусмотрен класс `ObservableCollection`, поддерживающий все необходимое для создания наблюдаемых коллекций. Также удачей следует считать обновление проекта `Fody` с целью автоматического добавления функциональности `IPropertyChanged`. При наличии под рукой упомянутых двух инструментов нет никаких причин отказываться от реализации наблюдаемых моделей в своих приложениях WPF.

Проверка достоверности WPF

Теперь, когда интерфейс `IPropertyChanged` реализован и задействован класс `ObservableCollection`, самое время заняться добавлением в приложение средств проверки достоверности. Приложениям необходимо проверять пользовательский ввод и обеспечивать обратную связь с пользователем, если введенные им данные оказываются некорректными. В настоящем разделе будут раскрыты наиболее распространенные механизмы проверки достоверности для современных приложений WPF, но это лишь часть возможностей, встроенных в инфраструктуру WPF.

Проверка достоверности происходит, когда привязка данных пытается обновить источник данных. В дополнение к встроенным проверкам, таким как исключения в блоках `set` для свойств, можно создавать специальные правила проверки достоверности. Если любое правило проверки достоверности (встроенное или специальное) нарушается, то в игру вступает класс `Validation`, который обсуждается позже в главе.

На заметку! В каждом разделе главы можно продолжить работу с проектом из предыдущего раздела или создать копию проекта, специально предназначенную для нового раздела. Всем последующим разделам соответствуют отдельные проекты, которые доступны в каталоге с кодом для настоящей главы внутри хранилища GitHub.

Модификация примера для демонстрации проверки достоверности

В каталоге для этой главы внутри хранилища GitHub новый проект (скопированный из предыдущего примера) называется `WpfValidations`. Если вы работаете с тем же самым проектом, созданным в предыдущем разделе, то при копировании в свой проект кода из примеров, приведенных в текущем разделе, просто должны обращать внимание на изменения пространств имен.

Класс Validation

Прежде чем добавлять проверку достоверности в проект, важно понять назначение класса `Validation`. Он входит в состав инфраструктуры проверки достоверности и предоставляет методы и присоединяемые свойства, которые могут применяться для отображения результатов проверки. При обработке ошибок проверки обычно используются три основных свойства класса `Validation`, кратко описанные в табл. 28.2. Они будут применяться далее в разделе.

Таблица 28.2. Основные члены класса `Validation`

Член	Описание
<code>HasError</code>	Присоединяемое свойство, которое указывает, что правило проверки достоверности где-то в процессе было нарушено
<code>Errors</code>	Коллекция всех активных объектов <code>ValidationError</code>
<code>ErrorTemplate</code>	Шаблон элемента управления, который становится видимым и декорирует связанный элемент, когда свойство <code>HasError</code> установлено в <code>true</code>

Варианты проверки достоверности

Как упоминалось ранее, технологии XAML поддерживают несколько механизмов для встраивания логики проверки достоверности внутрь приложения. В последующих разделах рассматриваются три самых распространенных варианта проверки.

Уведомление по исключениям

Хотя исключения не должны использоваться для обеспечения выполнения бизнес-логики, они могут (и будут) возникать, а потому требуют надлежащей обработки. Если исключения не обработаны в коде, тогда пользователь должен получить визуальную обратную связь об имеющейся проблеме. В отличие от Windows Forms в инфраструктуре WPF исключения привязки (по умолчанию) не распространяются до пользователя как собственно исключения. Тем не менее, они указываются визуально с применением декоратора (визуального уровня), который находится над элементами управления).

Запустите приложение, выберите запись в элементе ComboBox и очистите значение в текстовом поле Id. Поскольку свойство Id определено как имеющее тип int (не тип int, допускающий null), требуется числовое значение. После покидания поля Id по нажатию клавиши <Tab> механизм привязки отправляет свойству CarId пустую строку, но из-за того, что пустая строка не может быть преобразована в значение int, внутри блока set генерируется исключение. В нормальных обстоятельствах необработанное исключение привело бы к отображению окна сообщения пользователю, но в данном случае ничего подобного не происходит. Взглянув на порцию Debug (Отладка) окна Output (Вывод), вы заметите следующие строки:

```
System.Windows.Data Error: 7 : ConvertBack cannot convert value ''
(type 'String').
```

```
BindingExpression:Path=Id; DataItem='Car' (HashCode=52579650);
target element is 'TextBox' (Name=''); target property is 'Text'
(type 'String') FormatException:'System.
FormatException: Input string was not in a correct format.
```

Ошибка System.Windows.Data: 7 : ConvertBack не может преобразовать значение '' (типа String).

```
BindingExpression:Path=Id; DataItem='Car' (HashCode=52579650);
целевой элемент - TextBox (Name=''); целевое свойство - Text
(типа String) FormatException:'System.FormatException:
Входная строка не имела корректный формат.
```

Визуально исключение представляется с помощью тонкого прямоугольника красного цвета вокруг элемента управления (рис. 28.2).

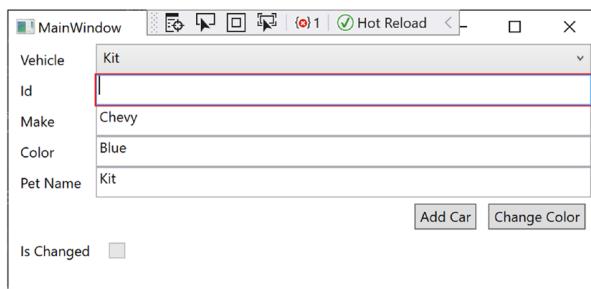


Рис. 28.2. Шаблон отображения ошибки по умолчанию

Прямоугольник красного цвета — это свойство `ErrorTemplate` объекта `Validation`, которое действует в качестве декоратора для связанного элемента управления. Несмотря на то что стандартный внешний вид говорит о наличии ошибки, нет никакого указания на то, что именно пошло не так. Хорошая новость в том, что шаблон отображения ошибки в свойстве `ErrorTemplate` является полностью настраиваемым, как вы увидите позже в главе.

Интерфейс `IDataErrorInfo`

Интерфейс `IDataErrorInfo` предоставляет механизм для добавления специальной проверки достоверности в классы моделей. Данный интерфейс добавляется прямо в классы моделей (или моделей представлений), а код проверки помещается внутри классов моделей (предпочтительно в частичные классы). Такой подход централизует код проверки достоверности в проекте, что совершенно не похоже на инфраструктуру Windows Forms, где проверка обычно делалась в самом пользовательском интерфейсе.

Показанный далее интерфейс `IDataErrorInfo` содержит два свойства: индексатор и строковое свойство по имени `Error`. Следует отметить, что механизм привязки WPF не задействует свойство `Error`.

```
public interface IDataErrorInfo
{
    string this[string columnName] { get; }
    string Error { get; }
}
```

Вскоре вы добавите частичный класс `Car`, но сначала необходимо модифицировать класс в файле `Car.cs`, пометив его как частичный. Добавьте в папку `Models` еще один файл по имени `CarPartial.cs`. Переименуйте этот класс в `Car`, пометьте его как `partial` и обеспечьте реализацию классом интерфейса `IDataErrorInfo`. Затем реализуйте члены интерфейса `IDataErrorInfo`. Вот начальный код:

```
public partial class Car : IDataErrorInfo
{
    public string this[string columnName] => string.Empty;
    public string Error { get; }
}
```

Чтобы привязанный элемент управления мог работать с интерфейсом `IDataErrorInfo`, в выражение привязки потребуется добавить `ValidatesOnDataErrors`. Модифицируйте выражение привязки для текстового поля `Make` следующим образом (и аналогично обновите остальные конструкции привязки):

```
<TextBox Grid.Column="1" Grid.Row="1"
         Text="{Binding Path=Make, ValidatesOnDataErrors=True}" />
```

После внесения изменений в конструкции привязки индексатор вызывается на модели каждый раз, когда возникает событие `PropertyChanged`. В качестве параметра `columnName` индексатора используется имя свойства из события. Если индексатор возвращает `string.Empty`, то инфраструктура предполагает, что все проверки достоверности прошли успешно и какие-либо ошибки отсутствуют. Если индексатор возвращает значение, отличающееся от `string.Empty`, тогда в свойстве для данного объекта присутствует ошибка, из-за чего каждый элемент управления, привязанный к этому свойству специфического экземпляра класса, считается содержащим ошибку.

Свойство HasError объекта Validation устанавливается в true и активизируется декоратором ErrorTemplate для элементов управления, на которые повлияла ошибка.

Добавьте простую логику проверки достоверности к индексатору в файле CorePartial.cs. Правила проверки элементарны:

- если Make равно ModelT, то установить сообщение об ошибке в "Too Old" (слишком старая модель);
- если Make равно Chevy и Color равно Pink, то установить сообщение об ошибке в "\${Make}'s don't come in {Color}" (модель в таком цвете не поставляется).

Начните с добавления оператора switch для каждого свойства. Во избежание применения "магических" строк в операторах case вы снова будете использовать операцию nameof. В случае сквозного прохода через оператор switch возвращается string.Empty. Далее добавьте правила проверки достоверности. В подходящих операторах case реализуйте проверку значения свойства на основе приведенных выше правил. В операторе case для свойства Make первым делом проверьте, равно ли значение ModelT. Если это так, тогда возвратите сообщение об ошибке. В случае успешного прохождения проверки в следующей строке кода вызовите вспомогательный метод, который возвратит сообщение об ошибке, если нарушено второе правило, или string.Empty, если нет. В операторе case для свойства Color просто вызовите тот же вспомогательный метод. Ниже показан код:

```
public string this[string columnName]
{
    get
    {
        switch (columnName)
        {
            case nameof(Id):
                break;
            case nameof(Make):
                return Make == "ModelT"
                    ? "Too Old"
                    : CheckMakeAndColor();
            case nameof(Color):
                return CheckMakeAndColor();
            case nameof(PetName):
                break;
        }
        return string.Empty;
    }
}

internal string CheckMakeAndColor()
{
    if (Make == "Chevy" && Color == "Pink")
    {
        return $"{Make}'s don't come in {Color}";
    }
    return string.Empty;
}
```

Запустите приложение, выберите автомобиль Red Rider (Ford) и измените значение в поле Make (Производитель) на ModelT. После того, как фокус покинет поле, появится декоратор ошибки красного цвета. Выберите в поле со списком автомобиль Kit (Chevy) и щелкните на кнопке Change Color, чтобы изменить его цвет на Pink. Вокруг поля Color немедленно появится декоратор ошибки красного цвета, но возле поля Make он будет отсутствовать. Измените значение в поле Make на Ford и переместите фокус из этого поля; декоратор ошибки красного цвета не появляется!

Причина в том, что индексатор выполняется, только когда для свойства генерировано событие PropertyChanged. Как обсуждалось в разделе "Система уведомлений привязки WPF" ранее в главе, событие PropertyChanged инициируется при изменении исходного значения свойства объекта, что происходит либо через код (вроде обработчика события Click для кнопки Change Color), либо через взаимодействие с пользователем (синхронизируется с помощью UpdateSourceTrigger). При изменении цвета свойство Make не изменяется, а потому событие PropertyChanged для него не генерируется. Поскольку событие не генерируется, индексатор не вызывается и проверка достоверности для свойства Make не выполняется.

Решить проблему можно двумя путями. Первый предусматривает изменение объекта PropertyChangedEventArgs, которое обеспечит обновление всех привязанных свойств, за счет передачи его конструктору значения string.Empty вместо имени поля. Как упоминалось ранее, это заставит механизм привязки обновить каждое свойство в данном экземпляре. Добавьте метод OnPropertyChanged() со следующим кодом:

```
protected virtual void OnPropertyChanged([CallerMemberName]
    string propertyName = "")
{
    if (propertyName != nameof(IsChanged))
    {
        IsChanged = true;
    }
    //PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(string.Empty));
}
```

Теперь при выполнении того же самого теста текстовые поля Make и Color декорируются с помощью шаблона отображения ошибки, когда одно из них обновляется. Так почему бы ни генерировать событие всегда в такой манере? В значительной степени причиной является производительность. Вполне возможно, что обновление каждого свойства объекта приведет к снижению производительности. Разумеется, без тестирования об этом утверждать нельзя, и конкретные ситуации могут (и вероятно будут) варьироваться.

Другое решение предполагает генерацию события PropertyChanged для зависимого поля (полей), когда одно из полей изменяется. Недостаток такого приема в том, что вы (или другие разработчики, сопровождающие ваше приложение) должны знать о взаимосвязи между свойствами Make и Color через код проверки достоверности.

Интерфейс INotifyDataErrorInfo

Интерфейс INotifyDataErrorInfo, появившийся в версии .NET 4.5, построен на основе интерфейса IDataErrorInfo и предлагает дополнительные возможности для проверки достоверности. Конечно, возросшая мощь сопровождается дополнительной работой! По разительному контрасту с предшествующими при-

емами проверки достоверности, которые вы видели до сих пор, свойство привязки `ValidatesOnNotifyDataErrors` имеет стандартное значение `true`, поэтому добавлять его к операторам привязки не обязательно.

Интерфейс `INotifyDataErrorInfo` чрезвычайно мал, но, как вскоре будет показано, для обеспечения своей эффективности требует написания порядочного объема связующего кода. Ниже приведено определение интерфейса `INotifyDataErrorInfo`:

```
public interface INotifyDataErrorInfo
{
    bool HasErrors { get; }
    event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
    IEnumerable GetErrors(string propertyName);
}
```

Свойство `HasErrors` используется механизмом привязки для выяснения, есть ли какие-нибудь ошибки в любых свойствах экземпляра. Если метод `GetErrors()` вызывается со значением `null` или пустой строкой в параметре `propertyName`, то он возвращает все ошибки, существующие в экземпляре. Если методу передан параметр `propertyName`, тогда возвращаются только ошибки, относящиеся к конкретному свойству. Событие `ErrorsChanged` (подобно событиям `PropertyChanged` и `CollectionChanged`) уведомляет механизм привязки о необходимости обновления пользовательского интерфейса для текущего списка ошибок.

Реализация поддерживающего кода

При реализации `INotifyDataErrorInfo` большая часть кода обычно помещается в базовый класс модели, поэтому она пишется только один раз. Начните с замены `IDataErrorInfo` интерфейсом `INotifyDataErrorInfo` в файле класса `CarPartial.cs` (код для `IDataErrorInfo` в классе можете оставить; вы обновите его позже).

```
public partial class Car: INotifyDataErrorInfo, IDataErrorInfo
{
    ...
    public IEnumerable GetErrors(string propertyName)
    {
        throw new NotImplementedException();
    }

    public bool HasErrors { get; }

    public event
        EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
}
```

Добавьте закрытое поле типа `Dictionary<string, List<string>>`, которое будет хранить сведения о любых ошибках, сгруппированные по именам свойств. Понадобится также добавить оператор `using` для пространства имён `System.Collections.Generic`. Вот как выглядит код:

```
using System.Collections.Generic;
private readonly Dictionary<string, List<string>> _errors =
    new Dictionary<string, List<string>>();
```

Свойство `HasErrors` должно возвращать `true`, если в словаре присутствуют любые ошибки, что легко достигается следующим образом:

```
public bool HasErrors => _errors.Any();
```

456 Часть VIII. Разработка клиентских приложений для Windows

Создайте вспомогательный метод для инициирования события ErrorsChanged (подобно инициированию события PropertyChanged):

```
private void OnErrorsChanged(string propertyName)
{
    ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
}
```

Как упоминалось ранее, метод GetErrors () должен возвращать любые ошибки в словаре, когда в параметре передается пустая строка или null. Если передается допустимое значение propertyName, то возвращаются ошибки, обнаруженные для указанного свойства. Если параметр не соответствует какому-либо свойству (или ошибки для свойства отсутствуют), тогда метод возвратит null.

```
public IEnumerable GetErrors(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName))
    {
        return _errors.Values;
    }
    return _errors.ContainsKey(propertyName) ? _errors[propertyName] : null;
}
```

Финальный набор вспомогательных методов будет добавлять одну или большее число ошибок для свойства либо очищать все ошибки для свойства (или всех свойств). Не следует забывать о вызове вспомогательного метода OnErrorsChanged () каждый раз, когда словарь изменяется.

```
private void AddError(string propertyName, string error)
{
    AddErrors(propertyName, new List<string> { error });
}
private void AddErrors(string propertyName, IList<string> errors)
{
    if (errors == null || !errors.Any())
    {
        return;
    }
    var changed = false;
    if (!_errors.ContainsKey(propertyName))
    {
        _errors.Add(propertyName, new List<string>());
        changed = true;
    }
    foreach (var err in errors)
    {
        if (_errors[propertyName].Contains(err)) continue;
        _errors[propertyName].Add(err);
        changed = true;
    }
    if (changed)
    {
        OnErrorsChanged(propertyName);
    }
}
```

```

protected void ClearErrors(string propertyName = "")
{
    if (string.IsNullOrEmpty(propertyName))
    {
        _errors.Clear();
    }
    else
    {
        _errors.Remove(propertyName);
    }
    OnErrorsChanged(propertyName);
}

```

Возникает вопрос: когда приведенный выше код активизируется? Механизм привязки прослушивает событие `ErrorsChanged` и обновляет пользовательский интерфейс, если в коллекции ошибок для выражения привязки возникает изменение. Но код проверки по-прежнему нуждается в триггере для запуска. Доступны два механизма, которые обсуждаются далее.

Использование интерфейса `INotifyDataErrorInfo` для проверки достоверности

Одним из мест выполнения проверки на предмет ошибок являются блоки `set` для свойств, как демонстрируется в показанном ниже примере, упрощенном до единственной проверки на равенство свойства `Make` значению `ModelT`:

```

public string Make
{
    get { return _make; }
    set
    {
        if (value == _make) return;
        _make = value;
        if (Make == "ModelT")
        {
            AddError(nameof(Make), "Too Old");
        }
        else
        {
            ClearErrors(nameof(Make));
        }
        OnPropertyChanged(nameof(Make));
        OnPropertyChanged(nameof(Color));
    }
}

```

Основная проблема такого подхода состоит в том, что вам приходится сочетать логику проверки достоверности с блоками `set` для свойств, что делает код труднее в чтении и сопровождении.

Комбинирование `IDataErrorInfo` с `INotifyDataErrorInfo` для проверки достоверности

В предыдущем разделе было показано, что реализацию интерфейса `IDataErrorInfo` можно добавить к частичному классу, т.е. обновлять блоки `set` не понадобится. Кроме того, индексатор автоматически вызывается при возникновении события `PropertyChanged` в свойстве. Комбинирование `IDataErrorInfo` и

INotifyDataColumnInfo предоставляет дополнительные возможности для проверки достоверности из INotifyDataColumnInfo, а также отделение от блоков set, обеспечиваемое IDataColumnInfo.

Цель применения IDataColumnInfo не в том, чтобы запускать проверку достоверности, а в том, чтобы гарантировать вызов кода проверки, который задействует INotifyDataColumnInfo, каждый раз, когда для объекта генерируется событие PropertyChanged. Поскольку интерфейс IDataColumnInfo не используется для проверки достоверности, необходимо всегда возвращать string.Empty из индексатора. Модифицируйте индексатор и вспомогательный метод CheckMakeAndColor() следующим образом:

```

public string this[string columnName]
{
    get
    {
        ClearErrors(columnName);
        switch (columnName)
        {
            case nameof(Id):
                break;
            case nameof(Make):
                CheckMakeAndColor();
                if (Make == "ModelT")
                {
                    AddError(nameof(Make), "Too Old");
                    hasError = true;
                }
                break;
            case nameof(Color):
                CheckMakeAndColor();
                break;
            case nameof(PetName):
                break;
        }
        return string.Empty;
    }
}
internal bool CheckMakeAndColor()
{
    if (Make == "Chevy" && Color == "Pink")
    {
        AddError(nameof(Make), $"{nameof(Make)}'s don't come in {Color}");
        AddError(nameof(Color),
            $"{nameof(Make)}'s don't come in {Color}");
        return true;
    }
    return false;
}

```

Запустите приложение, выберите автомобиль Chevy и измените цвет на Pink. В дополнение к декораторам красного цвета вокруг текстовых полей Make и Model будет также отображаться декоратор в виде красного прямоугольника, охватывающего целиком всю сетку, в которой находятся поля с детальной информацией об автомобиле (рис. 28.3).

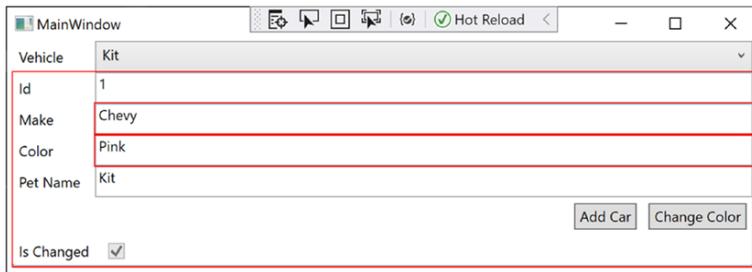


Рис. 28.3. Обновленный декоратор ошибки

Это еще одно преимущество применения интерфейса `INotifyDataErrorInfo`. В дополнение к элементам управления, которые содержат ошибки, элемент управления, определяющий контекст данных, также декорируется шаблоном отображения ошибки.

Отображение всех ошибок

Свойство `Errors` класса `Validation` возвращает все ошибки проверки достоверности для конкретного объекта в форме объектов `ValidationError`. Каждый объект `ValidationError` имеет свойство `ErrorContent`, которое содержит список сообщений об ошибках для свойства. Это означает, что сообщения об ошибках, которые нужно отобразить, находятся в списке внутри списка. Чтобы вывести их надлежащим образом, понадобится создать элемент `ListBox`, содержащий еще один элемент `ListBox`. Звучит слегка запутанно, но вскоре все прояснится.

Первым делом добавьте одну строку в `DetailsGrid` и удостоверьтесь в том, что значение свойства `Height` элемента `Window` составляет не менее 300. Поместите в последнюю строку элемент управления `ListBox` и привяжите его свойство `ItemsSource` к `DetailsGrid`, используя `Validation.Errors` для `Path`:

```
<ListBox Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
         ItemsSource="{Binding ElementName=DetailsGrid,
                               Path=(Validation.Errors)}">
</ListBox>
```

Добавьте к `ListBox` элемент `DataTemplate`, а в него — элемент управления `ListBox`, привязанный к свойству `ErrorContent`. Контекстом данных для каждого элемента `ListBoxItem` в этом случае является объект `ValidationError`, так что устанавливать контекст данных не придется, а только путь. Установите путь привязки в `ErrorContent`:

```
<ListBox.ItemTemplate>
<DataTemplate>
<ListBox ItemsSource="{Binding Path=ErrorContent}" />
</DataTemplate>
</ListBox.ItemTemplate>
```

Запустите приложение, выберите автомобиль `Chevy` и установите цвет в `Pink`. В окне отобразятся ошибки (рис. 28.4).

Мы лишь слегка коснулись поверхности того, что можно делать при проверке достоверности и отображении сообщений об ошибках, но представленных сведений должно быть вполне достаточно для выработки вами способа разработки информативных пользовательских интерфейсов, которые улучшают восприятие.

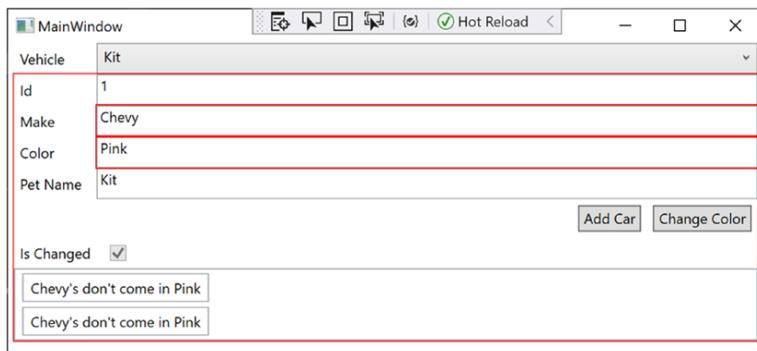


Рис. 28.4. Отображение коллекции ошибок

Перемещение поддерживающего кода в базовый класс

Вероятно, вы заметили, что в настоящий момент в классе `CarPartial` присутствует много кода. Поскольку в рассматриваемом примере есть только один класс модели, проблемы не возникают. Но по мере появления новых моделей в реальном приложении добавлять весь связующий код в каждый частичный класс для моделей нежелательно. Гораздо эффективнее поместить поддерживающий код в базовый класс, что и будет сделано.

Создайте в папке `Models` новый файл класса по имени `BaseEntity.cs`. Добавьте в него операторы `using` для пространств имен `System.Collections` и `System.ComponentModel`. Пометьте класс как открытый и обеспечьте реализацию им интерфейса `INotifyDataErrorInfo`.

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

namespace Validations.Models
{
    public class BaseEntity : INotifyDataErrorInfo
}
```

Переместите в новый базовый класс весь код, относящийся к `INotifyDataErrorInfo`, из файла `CarPartial.cs`. Любые закрытые методы понадобится сделать защищенными. Удалите реализацию интерфейса `INotifyDataErrorInfo` из класса в файле `CarPartial.cs` и добавьте `BaseEntity` в качестве базового класса:

```
public partial class Car : BaseEntity, IDataErrorInfo
{
    // Для краткости код не показан.
}
```

Теперь любые создаваемые классы моделей будут наследовать весь связующий код `INotifyDataErrorInfo`.

Использование аннотаций данных в WPF

Для проверки достоверности в пользовательских интерфейсах инфраструктура WPF способна также задействовать аннотации данных. Давайте добавим несколько аннотаций данных к модели Car.

Добавление аннотаций данных к модели

Откройте файл Car.cs и поместите в него оператор using для пространства имен System.ComponentModel.DataAnnotations. Добавьте к свойствам Make, Color и PetName атрибуты [Required] и [StringLength(50)]. Атрибут [Required] определяет правило проверки достоверности, которое регламентирует, что значение свойства не должно быть null (надо сказать, оно избыточно для свойства Id, т.к. свойство не относится к типу int, допускающему null). Атрибут [StringLength(50)] определяет правило проверки достоверности, которое ограничивает длину значения свойства 50 символами.

Контроль ошибок проверки достоверности на основе аннотаций данных

В WPF вы должны программно контролировать наличие ошибок проверки достоверности на основе аннотаций данных. Двумя основными классами, отвечающими за проверку достоверности на основе аннотаций данных, являются ValidationContext и Validator. Класс ValidationContext предоставляет контекст для контроля за наличием ошибок проверки достоверности. Класс Validator позволяет проверять, есть ли в объекте ошибки, связанные с аннотациями данных, в ValidationContext.

Откройте файл BaseEntity.cs и добавьте в него следующие операторы using:

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
```

Далее создайте новый метод по имени GetErrorsFromAnnotations(). Это обобщенный метод, который принимает в качестве параметров строковое имя свойства и значение типа T, а возвращает строковый массив. Он должен быть помечен как protected. Вот его сигнатура:

```
protected string[] GetErrorsFromAnnotations<T>(string propertyName, T value)
{
}
```

Внутри метода GetErrorsFromAnnotations() создайте переменную типа List<ValidationResult>, которая будет хранить результаты выполненных проверок достоверности, и объект ValidationContext с областью действия, ограниченной именем переданного методу свойства. Затем вызовите метод Validate.TryValidateProperty(), который возвращает значение bool. Если все проверки (на основе аннотаций данных) прошли успешно, тогда метод возвращает true. В противном случае он вернет false и наполнит List<ValidationResult> информацией о возникших ошибках. Полный код выглядит так:

```
protected string[] GetErrorsFromAnnotations<T>(
    string propertyName, T value)
{
    var results = new List<ValidationResult>();
    var vc = new ValidationContext(this, null, null)
    { MemberName = propertyName };
```

```

var isValid = Validator.TryValidateProperty(
    value, vc, results);
return (isValid)
    ? null
    : Array.ConvertAll(
        results.ToArray(), o => o.ErrorMessage);
}

```

Теперь можете модифицировать метод индексатора в файле CarPartial.cs, чтобы проверять наличие любых ошибок, основанных на аннотациях данных. Обнаруженные ошибки должны добавляться в коллекцию ошибок, поддерживаемую интерфейсом INotifyDataErrorInfo. Это позволяет привести в порядок обработку ошибок. В начале индексаторного метода очистите ошибки для столбца. Затем обработайте результаты проверок достоверности и в заключение предоставьте специальную логику для сущности. Ниже показан обновленный код индексатора:

```

public string this[string columnName]
{
    get
    {
        ClearErrors(columnName);
        var errorsFromAnnotations =
            GetErrorsFromAnnotations(columnName,
                typeof(Car)
                    .GetProperty(columnName)?.GetValue(this, null));
        if (errorsFromAnnotations != null)
        {
            AddErrors(columnName, errorsFromAnnotations);
        }
        switch (columnName)
        {
            case nameof(Id):
                break;
            case nameof(Make):
                CheckMakeAndColor();
                if (Make == "ModelT")
                {
                    AddError(nameof(Make), "Too Old");
                }
                break;
            case nameof(Color):
                CheckMakeAndColor();
                break;
            case nameof(PetName):
                break;
        }
        return string.Empty;
    }
}

```

Запустите приложение, выберите один из автомобилей и введите в поле Color текст, содержащий более 50 символов. После превышения порога в 50 символов аннотация данных StringLength создает ошибку проверки достоверности, которая сообщается пользователю (рис. 28.5).

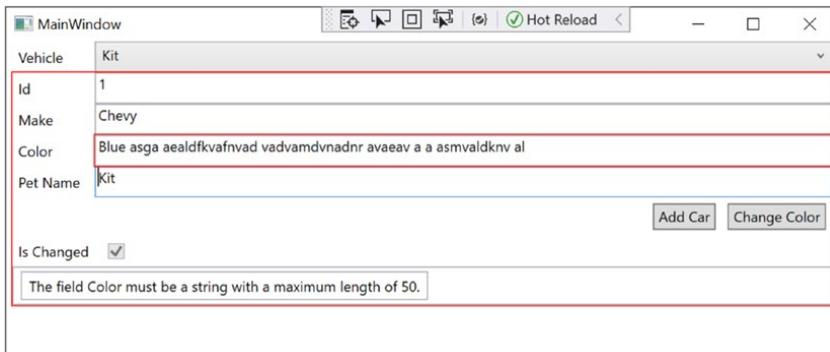


Рис. 28.5. Проверка достоверности на основе аннотаций данных

Настройка свойства ErrorTemplate

Финальной темой является создание стиля, который будет применяться, когда элемент управления содержит ошибку, а также обновление ErrorTemplate для отображения более осмысленного сообщения об ошибке. Как объяснялось в главе 27, элементы управления допускают настройку посредством стилей и шаблонов элементов управления.

Начните с добавления в раздел Window.Resources файла MainWindow.xaml нового стиля с целевым типом TextBox. Добавьте к стилю триггер, который устанавливает свойства, когда свойство Validation.HasError имеет значение true. Свойствами и устанавливаемыми значениями являются Background (Pink), Foreground (Black) и ToolTip (ErrorContent). В элементах Setter для свойств Background и Foreground нет ничего нового, но синтаксис установки свойства ToolTip требует пояснения. Привязка (Binding) указывает на элемент управления, к которому применяется данный стиль, в этом случае TextBox. Путь (Path) представляет собой первое значение ErrorContent в коллекции Validation.Errors. Разметка выглядит следующим образом:

```
<Window.Resources>
<Style TargetType="{x:Type TextBox}">
<Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
        <Setter Property="Background" Value="Pink" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="ToolTip"
            Value="{Binding RelativeSource={RelativeSource Self},
            Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
</Style.Triggers>
</Style>
</Window.Resources>
```

Запустите приложение и создайте условие для ошибки. Результат будет подобен тому, что показан на рис. 28.6, и укомплектован всплывающей подсказкой с сообщением об ошибке.

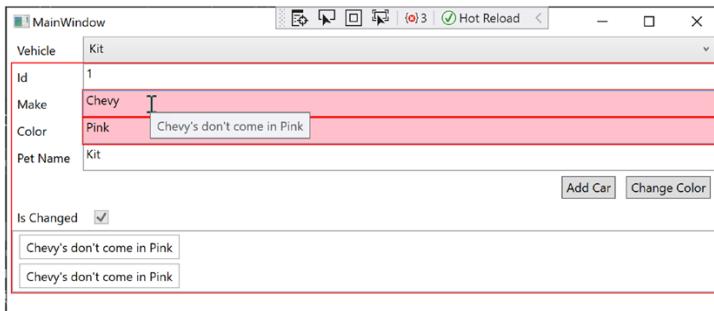


Рис. 28.6. Отображение специального шаблона ErrorTemplate

Определенный выше стиль изменяет внешний вид любого элемента управления TextBox, который содержит ошибку. Далее мы создадим специальный шаблон элемента управления с целью обновления свойства ErrorTemplate класса Validation, чтобы отобразить восклицательный знак красного цвета и установить всплывающие подсказки для восклицательного знака. Шаблон ErrorTemplate является декоратором, который располагается поверх элемента управления. Хотя только что созданный стиль обновляет сам элемент управления, шаблон ErrorTemplate будет размещаться поверх элемента управления.

Поместите элемент Setter непосредственно после закрывающего дескриптора Style.Triggers внутри созданного стиля. Вы будете создавать шаблон элемента управления, состоящий из элемента TextBlock (для отображения восклицательного знака) и элемента BorderBrush, который окружает TextBox, содержащий сообщение об ошибке (или несколько сообщений). В языке XAML предусмотрен специальный дескриптор для элемента управления, декорированного с помощью ErrorTemplate, под названием AdornedElementPlaceholder. Добавляя имя такого элемента управления, можно получить доступ к ошибкам, которые ассоциированы с элементом управления. В рассматриваемом примере вас интересует доступ к свойству Validation.Errors с целью получения ErrorContent (как делалось в Style.Trigger). Вот полная разметка для элемента Setter:

```
<Setter Property="Validation.ErrorTemplate">
<Setter.Value>
<ControlTemplate>
<DockPanel LastChildFill="True">
<TextBlock Foreground="Red" FontSize="20" Text="!" ToolTip="{Binding ElementName=controlWithError, Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"/>
<Border BorderBrush="Red" BorderThickness="1">
<AdornedElementPlaceholder Name="controlWithError" />
</Border>
</DockPanel>
</ControlTemplate>
</Setter.Value>
</Setter>
```

Запустите приложение и создайте условие для возникновения ошибки. Результат будет подобен представленному на рис. 28.7.

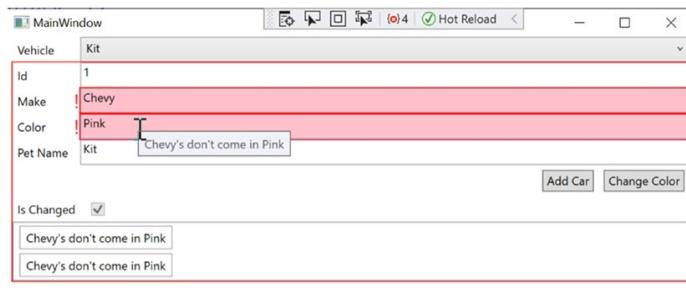


Рис. 28.7. Отображение обновленного специального шаблона ErrorTemplate

Итоговые сведения о проверке достоверности

На этом исследование методов проверки достоверности в WPF завершено. Разумеется, с их помощью можно делать намного большее. За дополнительными сведениями обращайтесь в документацию по WPF.

Создание специальных команд

Как и в разделе, посвященном проверке достоверности, можете продолжить работу с тем же проектом или создать новый проект и скопировать в него весь код из предыдущего проекта. Вы создадите новый проект по имени WpfCommands. В случае работы с проектом из предыдущего раздела обращайте внимание на пространства имен в примерах кода и корректируйте их по мере необходимости.

В главе 25 объяснялось, что команды являются неотъемлемой частью WPF. Команды могут привязываться к элементам управления WPF (таким как Button и MenuItem) для обработки пользовательских событий, подобных щелчку. Вместо создания обработчика события напрямую и помещения его кода в файл отдельного кода при возникновении события выполняется метод Execute() команды. Метод CanExecute() используется для включения или отключения элемента управления на основе специального кода. В дополнение к встроенным командам, которые применялись в главе 25, можно создавать собственные команды, реализуя интерфейс ICommand. Когда вместо обработчиков событий используются команды, появляются преимущества инкапсуляции кода приложения, а также автоматического включения и отключения элементов управления с помощью бизнес-логики.

Реализация интерфейса ICommand

Как было показано в главе 25, интерфейс ICommand определен следующим образом:

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;
    bool CanExecute(object parameter);
    void Execute(object parameter);
}
```

Добавление класса ChangeColorCommand

Обработчики событий для элементов управления Button вы замените командами, начав с кнопки Change Color. Создайте в проекте новую папку по имени Cmds. Добавьте в нее новый файл класса ChangeColorCommand.cs. Сделайте класс открытым и реализующим интерфейс ICommand. Добавьте приведенные ниже операторы using (первый может варьироваться в зависимости от того, создавался ли новый проект для данного примера):

```
using WpfCommands.Models;
using System.Windows.Input;
```

Код класса должен выглядеть примерно так:

```
public class ChangeColorCommand : ICommand
{
    public bool CanExecute(object parameter)
    {
        throw new NotImplementedException();
    }

    public void Execute(object parameter)
    {
        throw new NotImplementedException();
    }

    public event EventHandler CanExecuteChanged;
}
```

Если метод CanExecute() возвращает true, то привязанные элементы управления будут включенными, а если false, тогда они будут отключенными. Если элемент управления включен (CanExecute() возвращает true) и на нем совершается щелчок, то запустится метод Execute(). Параметры, передаваемые обоим методам, поступают из пользовательского интерфейса и основаны на свойстве CommandParameter, устанавливаемом в конструкциях привязки. Событие CanExecuteChanged предусмотрено в системе привязки и уведомлений для информирования пользовательского интерфейса о том, что результат, возвращаемый методом CanExecute(), изменился (почти как событие PropertyChanged).

В текущем примере кнопка Change Color должна работать, только если параметр отличается от null и принадлежит типу Car. Модифицируйте метод CanExecute() следующим образом:

```
public bool CanExecute(object parameter)
=> (parameter as Car) != null;
```

Значение параметра для метода Execute() будет таким же, как и для метода CanExecute(). Поскольку метод Execute() может выполняться лишь в случае, если объект имеет тип Car, аргумент потребуется привести к типу Car и затем обновить значение цвета:

```
public void Execute(object parameter)
{
    ((Car)parameter).Color="Pink";
}
```

Присоединение команды к *CommandManager*

Финальное обновление класса команды связано с присоединением команды к диспетчеру команд (*CommandManager*). Метод *CanExecute()* запускается при загрузке окна в первый раз и затем в ситуации, когда диспетчер команд инструктирует его о необходимости перезапуска. Каждый класс команды должен быть присоединен к диспетчеру команд, для чего нужно модифицировать код, относящийся к событию *CanExecuteChanged*:

```
public event EventHandler CanExecuteChanged
{
    add => CommandManager.RequerySuggested += value;
    remove => CommandManager.RequerySuggested -= value;
}
```

Изменение файла *MainWindow.xaml.cs*

Следующее изменение связано с созданием экземпляра класса *ChangeColorCommand*, к которому может иметь доступ элемент управления *Button*. В настоящий момент вы будете делать это в файле отделенного кода для *MainWindow* (позже в главе код переместится в модель представления). Откройте файл *MainWindow.xaml.cs* и удалите обработчик события *Click* для кнопки *Change Color*. Поместите в начало файла следующие операторы *using* (пространство имен может варьироваться в зависимости от того, работаете вы с предыдущим проектом или начали новый):

```
using WpfCommands.Cmds;
using System.Windows.Input;
```

Добавьте открытое свойство по имени *ChangeColorCmd* типа *ICommand* с поддерживающим полем. В теле выражения для свойства возвратите значение поддерживающего поля (создавая экземпляр *ChangeColorCommand*, если поддерживающее поле равно *null*):

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd
    => _changeColorCommand ??= new ChangeColorCommand();
```

Изменение файла *MainWindow.xaml*

Как было показано в главе 25, элементы управления WPF, реагирующие на щелчки (вроде *Button*), имеют свойство *Command*, которое позволяет назначать элементу управления объект команды. Для начала присоедините объект команды, созданный в файле отделенного кода, к кнопке *btnChangeColor*. Поскольку свойство для команды находится в классе *MainWindow*, с помощью синтаксиса привязки *RelativeSource* получается окно, содержащее необходимую кнопку:

```
Command="{Binding Path=ChangeColorCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
    AncestorType={x:Type Window}}}"
```

Кнопка также нуждается в передаче объекта *Car* в качестве параметра для методов *CanExecute()* и *Execute()*, что делается через свойство *CommandParameter*. Установите свойство *Path* для *CommandParameter* в свойство *SelectedItem* элемента *ComboBox* по имени *cboCars*:

```
CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"
```

Вот завершенная разметка для кнопки:

```
<Button x:Name="btnChangeColor" Content="Change Color" Margin="5,0,5,0"
    Padding="4, 2" Command="{Binding Path=ChangeColorCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
        AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Тестирование приложения

Запустите приложение. Кнопка Change Color не будет доступной (рис. 28.8), т.к. автомобиль еще не выбран.

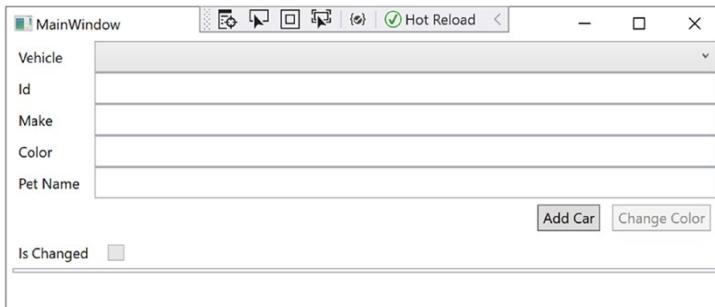


Рис. 28.8. Окно, где ничего не выбрано

Теперь выберите автомобиль; кнопка Change Color становится доступной, а щелчок на ней обеспечивает изменение цвета, как и ожидалось!

Создание класса CommandBase

Если распространить такой шаблон на AddCarCommand.cs, то итогом стал бы код, повторяющийся среди классов. Это хороший знак о том, что необходим базовый класс. Создайте внутри папки Cmds новый файл класса по имени CommandBase.cs и добавьте оператор using для пространства имен System.Windows.Input. Сделайте класс CommandBase открытым и реализующим интерфейс ICommand. Превратите класс и методы Execute() и CanExecute() в абстрактные. Наконец, добавьте обновление в событие CanExecuteChanged из класса ChangeColorCommand. Ниже показана полная реализация:

```
using System;
using System.Windows.Input;
namespace WpfCommands.Cmds
{
    public abstract class CommandBase : ICommand
    {
        public abstract bool CanExecute(object parameter);
        public abstract void Execute(object parameter);
        public event EventHandler CanExecuteChanged
        {
            add => CommandManager.RequerySuggested += value;
            remove => CommandManager.RequerySuggested -= value;
        }
    }
}
```

Добавление класса AddCarCommand

Добавьте в папку Cmds новый файл класса по имени AddCarCommand.cs. Сделайте класс открытым и укажите CommandBase в качестве базового класса. Поместите в начало файла следующие операторы using:

```
using System.Collections.ObjectModel;
using System.Linq;
using WpfCommands.Models;
```

Ожидается, что параметр должен иметь тип ObservableCollection<Car>, поэтому предусмотрите в методе CanExecute() соответствующую проверку. Если параметр относится к типу ObservableCollection<Car>, тогда метод Execute() должен добавить дополнительный объект Car подобно обработчику события Click.

```
public class AddCarCommand : CommandBase
{
    public override bool CanExecute(object parameter)
        => parameter is ObservableCollection<Car>;
    public override void Execute(object parameter)
    {
        if (parameter is not ObservableCollection<Car> cars)
        {
            return;
        }
        var maxCount = cars.Max(x => x.Id);
        cars.Add(new Car
        {
            Id = ++maxCount,
            Color = "Yellow",
            Make = "VW",
            PetName = "Birdie"
        });
    }
}
```

Изменение файла MainWindow.xaml.cs

Добавьте открытое свойство типа ICommand по имени AddCarCmd с поддерживающим полем. В теле выражения для свойства возвратите значение поддерживающего поля (создавая экземпляр AddCarCommand, если поддерживающее поле равно null):

```
private ICommand _addCarCommand = null;
public ICommand AddCarCmd
    => _addCarCommand ??= new AddCarCommand();
```

Изменение файла MainWindow.xaml

Модифицируйте разметку XAML, удалив атрибут Click и добавив атрибуты Command и CommandParameter. Объект AddCarCommand будет получать список автомобилей из поля со списком cboCars. Ниже показана полная разметка XAML для кнопки:

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=AddCarCmd,
    RelativeSource={RelativeSource Mode=FindAncestor,
    AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=ItemsSource}" />
```

В результате появляется возможность добавления автомобилей и обновления их цветов (пока с весьма ограниченной функциональностью) с помощью многократно используемого кода, содержащегося в автономных классах.

Изменение класса *ChangeColorCommand*

Финальным шагом будет обновление класса ChangeColorCommand, чтобы он стал унаследованным от CommandBase. Замените интерфейс ICommand классом CommandBase, добавьте к обоим методам ключевое слово override и удалите код события CanExecuteChanged. Все оказалось действительно настолько просто! Вот как выглядит новый код:

```
public class ChangeColorCommand : CommandBase
{
    public override bool CanExecute(object parameter)
        => parameter is Car;
    public override void Execute(object parameter)
    {
        ((Car)parameter).Color = "Pink";
    }
}
```

Объекты *RelayCommand*

Еще одной реализацией паттерна “Команда” (Command) в WPF является RelayCommand. Вместо создания нового класса, представляющего каждую команду, данный паттерн применяет делегаты для реализации интерфейса ICommand. Реализация легковесна в том, что каждая команда не имеет собственного класса. Объекты RelayCommand обычно используются, когда нет необходимости в многократном применении реализации команды.

Создание базового класса *RelayCommand*

Как правило, объекты RelayCommand реализуются в двух классах. Базовый класс RelayCommand используется при отсутствии каких-либо параметров для методов CanExecute() и Execute(), а класс RelayCommand<T> применяется, когда требуется параметр. Начните с базового класса RelayCommand, который задействует класс CommandBase. Добавьте в папку Cmds новый файл класса по имени RelayCommand.cs. Сделайте его открытым и укажите CommandBase в качестве базового класса. Добавьте две переменные уровня класса для хранения делегатов Execute() и CanExecute():

```
private readonly Action _execute;
private readonly Func<bool> _canExecute;
```

Создайте три конструктора. Первый — стандартный конструктор (необходимый для производного класса RelayCommand<T>), второй — конструктор, который принимает параметр Action, и третий — конструктор, принимающий параметры Action и Func:

```
public RelayCommand() { }
public RelayCommand(Action execute) : this(execute, null) { }
public RelayCommand(Action execute, Func<bool> canExecute)
{
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
}
```

Наконец, реализуйте переопределенные версии `CanExecute()` и `Execute()`. Метод `CanExecute()` возвращает `true`, если параметр `Func` равен `null`; если же параметр `Func` не `null`, то он выполняется и возвращается `true`. Метод `Execute()` выполняет параметр типа `Action`.

```
public override bool CanExecute(object parameter)
    => _canExecute == null || _canExecute();
public override void Execute(object parameter) { _execute(); }
```

Создание класса `RelayCommand<T>`

Добавьте в папку `Cmds` новый файл класса по имени `RelayCommandT.cs`. Класс `RelayCommandT` является почти полной копией базового класса, исключая тот факт, что все делегаты принимают параметр. Сделайте класс открытым и обобщенным, а также унаследованным от базового класса `RelayCommand`:

```
public class RelayCommand<T> : RelayCommand
```

Добавьте две переменные уровня класса для хранения делегатов `Execute()` и `CanExecute()`:

```
private readonly Action<T> _execute;
private readonly Func<T, bool> _canExecute;
```

Создайте два конструктора. Первый из них принимает параметр `Action<T>`, а второй — параметры `Action<T>` и `Func<T, bool>`:

```
public RelayCommand(Action<T> execute) : this(execute, null) { }
public RelayCommand(Action<T> execute, Func<T, bool> canExecute)
{
    _execute = execute ?? throw new ArgumentNullException(nameof(execute));
    _canExecute = canExecute;
}
```

Наконец, реализуйте переопределенные версии `CanExecute()` и `Execute()`. Метод `CanExecute()` возвращает `true`, если `Func` равно `null`, а иначе выполняет `Func` и возвращает `true`. Метод `Execute()` выполняет параметр типа `Action`.

```
public override bool CanExecute(object parameter)
    => _canExecute == null || _canExecute((T)parameter);
public override void Execute(object parameter)
    { _execute((T)parameter); }
```

Изменение файла `MainWindow.xaml.cs`

Когда используются объекты `RelayCommand`, при конструировании новой команды должны указываться все методы для делегатов. Это вовсе не означает, что код нуждается в помещении внутрь файла отделенного кода (как показано здесь); он просто должен быть доступным из файла отделенного кода. Код может находиться в другом классе (или даже в другой сборке), что дает преимущества инкапсуляции, связанные с созданием специального класса команды.

Добавьте новую закрытую переменную типа `RelayCommand<Car>` и открытое свойство по имени `DeleteCarCmd`:

```
private RelayCommand<Car> _deleteCarCommand = null;
public RelayCommand<Car> DeleteCarCmd
    => _deleteCarCommand ??=
        new RelayCommand<Car>(DeleteCar, CanDeleteCar));
```

Также потребуется создать методы `DeleteCar()` и `CanDeleteCar()`:

```
private bool CanDeleteCar(Car car) => car != null;
private void DeleteCar(Car car)
{
    _cars.Remove(car);
}
```

Обратите внимание на строгую типизацию в методах — одно из преимуществ применения `RelayCommand<T>`.

Добавление и реализация кнопки удаления записи об автомобиле

Последним шагом будет добавление кнопки `Delete Car` (Удалить автомобиль) и установка привязок `Command` и `CommandParameter`. Добавьте следующую разметку:

```
<Button x:Name="btnDeleteCar" Content="Delete Car"
Margin="5,0,5,0" Padding="4, 2"
Command="{Binding Path=DeleteCarCmd,
RelativeSource={RelativeSource Mode=FindAncestor,
AncestorType={x:Type Window}}}"
CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

Теперь, запустив приложение, вы можете удостовериться в том, что кнопка `Delete Car` доступна, только если в раскрывающемся списке выбран автомобиль, и щелчок на ней приводит к удалению записи об автомобиле.

Итоговые сведения о команде

На этом краткий экскурс в команды WPF завершен. За счет перемещения кода обработки событий из файла отдельного кода в индивидуальные классы команд появляются преимущества инкапсуляции, многократного использования и улучшенной возможности сопровождения. Если настолько большое разделение обязанностей не требуется, тогда можно применять легковесную реализацию `RelayCommand`. Цель заключается в том, чтобы улучшить возможность сопровождения и качество кода, так что выбирайте подход, который лучше подходит для вашей ситуации.

Перенос кода и данных в модель представления

Как и в разделе “Проверка достоверности WPF”, вы можете продолжить работу с тем же самым проектом или создать новый и скопировать в него весь код. Вы создадите новый проект по имени `WpfViewModel`. В случае работы с проектом из предыдущего раздела обращайте внимание на пространства имен в примерах кода и корректируйте их по мере необходимости.

Создайте в проекте новую папку под названием `ViewModels` и поместите в нее новый файл класса `MainWindowViewModel.cs`. Добавьте операторы `using` для следующих пространств имен и сделайте класс открытым:

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Windows.Input;
using WpfViewModel.Cmnds;
using WpfViewModel.Models;
```

На заметку! Популярное соглашение предусматривает именование моделей представлений в соответствие с окном, которое их поддерживает. Обычно имеет смысл следовать такому соглашению, поэтому оно соблюдается в настоящей главе. Тем не менее, как и любой паттерн или соглашение, это не норма, и на данный счет вы найдете широкий спектр мнений.

Перенос кода MainWindow.xaml.cs

В модель представления будет перемещен почти весь код из файла отделенного кода. В конце останется только несколько строк, включая вызов метода `InitializeComponent()` и код установки контекста данных для окна в модель представления.

Создайте открытое свойство типа `IList<Car>` по имени `Cars`:

```
public IList<Car> Cars { get; } =
    new ObservableCollection<Car>();
```

Создайте стандартный конструктор и перенесите в него весь код создания объектов `Car` из файла `MainWindow.xaml.cs`, обновив имя списковой переменной. Можете также удалить переменную `_cars` из `MainWindow.xaml.cs`. Ниже показан конструктор модели представления:

```
public MainWindowViewModel()
{
    Cars.Add(
        new Car { Id = 1, Color = "Blue", Make = "Chevy",
            PetName = "Kit", IsChanged = false });
    Cars.Add(
        new Car { Id = 2, Color = "Red", Make = "Ford",
            PetName = "Red Rider", IsChanged = false });
}
```

Далее переместите весь код, относящийся к командам, из файла отделенного кода окна в модель представления, заменив ссылку на переменную `_cars` ссылкой на `Cars`. Вот измененный код:

```
// Для краткости остальной код не показан.
private void DeleteCar(Car car)
{
    Cars.Remove(car);
}
```

Обновление кода и разметки MainWindow

Из файла `MainWindow.xaml.cs` кода была удалена большая часть кода. Удалите строку, которая устанавливает `ItemsSource` для поля со списком, оставив только вызов `InitializeComponent()`. Код должен выглядеть примерно так:

```
public MainWindow()
{
    InitializeComponent();
}
```

Добавьте в начало файла следующий оператор `using`:

```
using WpfViewModel.ViewModels;
```

Создайте строго типизированное свойство для хранения экземпляра модели представления:

```
public MainWindowViewModel ViewModel { get; set; } =
    new MainWindowViewModel();
```

Добавьте свойство DataContext к объявлению окна в разметке XAML:

```
DataContext="{Binding ViewModel, RelativeSource={RelativeSource Self}}"
```

Обновление разметки элементов управления

Теперь, когда свойство DataContext для Window установлено в модель представления, потребуется обновить привязки элементов управления в разметке XAML. Начиная с поля со списком, модифицируйте разметку за счет добавления свойства ItemsSource:

```
<ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName"
    ItemsSource="{Binding Cars}" />
```

Прием работает, т.к. контекстом данных для Window является MainWindowViewModel, а Cars — открытое свойство модели представления. Вспомните, что конструкции привязки обходят дерево элементов до тех пор, пока не найдут контекст данных. Далее понадобится обновить привязки для элементов управления Button. Задача проста; поскольку привязки уже установлены на уровне окна, нужно лишь модифицировать конструкции привязки, чтобы они начинались со свойства DataContext:

```
<Button x:Name="btnAddCar" Content="Add Car"
    Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.AddCarCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars,
        Path=ItemsSource}"/>
<Button x:Name="btnDeleteCar" Content="Delete Car"
    Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.DeleteCarCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars,
        Path=SelectedItem}"/>
<Button x:Name="btnChangeColor" Content="Change Color"
    Margin="5,0,5,0" Padding="4, 2"
    Command="{Binding Path=DataContext.ChangeColorCmd,
        RelativeSource={RelativeSource Mode=FindAncestor,
            AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars,
        Path=SelectedItem}"/>
```

Итоговые сведения о моделях представлений

Верите или нет, но вы только что закончили построение первого WPF-приложения MVVM. Вы можете подумать: “Это не реалистичное приложение. Как насчет данных? Данные в примере жестко закодированы”. И вы будете совершенно правы. Это не реальное приложение, а лишь демонстрация. Однако в ней легко оценить всю прелест паттерна MVVM. Представлению ничего не известно о том, откуда поступают данные;

оно просто привязывается к свойству модели представления. Реализации модели представления можно менять, скажем, использовать версию с жестко закодированными данными во время тестирования и версию, обращающуюся к базе данных, в производственной среде.

Можно было бы обсудить еще немало вопросов, в том числе разнообразные инфраструктуры с открытым кодом, паттерн "Локатор модели представления" (View Model Locator) и множество разных мнений на предмет того, как лучше реализовывать паттерн MVVM. В том и заключается достоинство паттернов проектирования программного обеспечения — обычно существует много правильных способов их реализации, и вам необходимо лишь отыскать стиль, который наилучшим образом подходит к имеющимся требованиям.

Обновление проекта AutoLot.Dal для MVVM

Если вы хотите обновить проект AutoLot.Dal для MVVM, то должны применить изменения, которые вносились в Car, ко всем сущностям в проекте AutoLot.Dal.Models, включая BaseEntity.

Резюме

В главе рассматривались аспекты WPF, относящиеся к поддержке паттерна MVVM. Сначала было показано, каким образом связывать классы моделей и коллекции с помощью системы уведомлений в диспетчере привязки. Демонстрировалась реализация интерфейса INotifyPropertyChanged и применение классов наблюдаемых коллекций для обеспечения синхронизации пользовательского интерфейса и связанных с ним данных.

Вы научились добавлять код проверки достоверности к модели с применением интерфейсов IDataErrorInfo и INotifyDataErrorInfo, а также проверять наличие ошибок, основанных на аннотациях данных. Было показано, как отображать обнаруженные ошибки проверки достоверности в пользовательском интерфейсе, чтобы пользователь знал о проблеме и мог ее устраниТЬ. Вдобавок был создан стиль и специальный шаблон элементов управления для визуализации ошибок более эффективным способом.

В заключение вы узнали, каким образом собирать все компоненты вместе за счет добавления модели представления, а также приводить в порядок разметку и отделенный код пользовательского интерфейса, чтобы усилить разделение обязанностей.

ЧАСТЬ IX

ASP.NET Core

ГЛАВА 29

Введение в ASP.NET Core

В финальной части книги рассматривается ASP.NET Core — последняя версия инфраструктуры для разработки веб-приложений, которая использует C# и .NET Core. В этой главе предлагается введение в инфраструктуру ASP.NET Core и раскрываются ее отличия от предыдущей версии, т.е. ASP.NET.

После ознакомления с основами паттерна “модель–представление–контроллер” (Model-View-Controller — MVC), реализованного в ASP.NET Core, вы приступите к построению двух приложений, которые будут работать вместе. Первое приложение, REST-служба ASP.NET Core, будет закончено в главе 30. Вторым приложением является веб-приложение ASP.NET Core, созданное с применением паттерна MVC, которое будет завершено в главе 31. Уровнем доступа к данным для обоих приложений послужат проекты AutoLot.Dal и AutoLot.Models, которые вы создали в главе 23.

Краткий экскурс в прошлое

Выпуск инфраструктуры ASP.NET MVC в 2007 году принес большой успех компании Microsoft. Инфраструктура базировалась на паттерне MVC и стала ответом разработчикам, разочарованным API-интерфейсом Web Forms, который по существу был ненадежной абстракцией поверх HTTP. Инфраструктура Web Forms проектировалась для того, чтобы помочь разработчикам клиент-серверных приложений перейти к созданию веб-приложений, и в этом отношении она была довольно успешной. Однако по мере того, как разработчики все больше и больше привыкали к процессу разработки веб-приложений, многим из них хотелось иметь более высокую степень контроля над визуализируемым выводом, избавиться от состояния представления и ближе придерживаться проверенных паттернов проектирования для веб-приложений. С учетом указанных целей и создавалась инфраструктура ASP.NET MVC.

Введение в паттерн MVC

Паттерн “модель–представление–контроллер” (Model-View-Controller — MVC) появился в 1970-х годах, будучи первоначально созданным для использования в Smalltalk. Относительно недавно его популярность возросла, в результате чего стали доступными реализации в различных языках, в том числе Java (Spring Framework), Ruby (Ruby on Rails) и .NET (ASP.NET MVC).

Модель

Модель — это данные в приложении. Данные обычно представляются с помощью простых старых объектов CLR (plain old CLR object — РОСО). Модели представлений состоят из одной или большего числа моделей и приспособлены специально для пот-

ребителя данных. Воспринимайте модели и модели представлений как таблицы базы данных и представления базы данных.

С академической точки зрения модели должны быть в высшей степени чистыми и не содержать правила проверки достоверности или любые другие бизнес-правила. С практической точки зрения тот факт, содержит модель логику проверки достоверности или другие бизнес-правила, целиком зависит от применяемого языка и инфраструктур, а также специфических потребностей приложения. Например, в инфраструктуре EF Core присутствует много аннотаций данных, которые имеют двойное назначение: механизм для формирования таблиц базы данных и средство для проверки достоверности в веб-приложениях ASP.NET Core. Примеры, приводимые в книге, сконцентрированы на сокращении дублированного кода, что приводит к размещению аннотаций данных и проверок достоверности там, где в них есть наибольший смысл.

Представление

Представление — это пользовательский интерфейс приложения. Представление принимает команды и визуализирует результаты команд для пользователя. Представление обязано быть как можно более легковесным и не выполнять какую-то фактическую работу, а передавать всю работу контроллеру.

Контроллер

Контроллер является своего рода мозговым центром функционирования. Контроллеры принимают команды/запросы от пользователя (через представления) или клиента (через обращения к API-интерфейсу) посредством методов действий и надлежащим образом их обрабатывают. Результат операции затем возвращается пользователю или клиенту. Контроллеры должны быть легковесными и использовать другие компоненты или службы для обработки запросов, что содействует разделению обязанностей и улучшает возможности тестирования и сопровождения.

ASP.NET Core и паттерн MVC

С помощью ASP.NET Core можно создавать много типов веб-приложений и служб. Двумя вариантами являются веб-приложения, в которых применяются паттерн MVC и службы REST. Если вы имели дело с “классической” инфраструктурой ASP.NET, то знайте, что они аналогичны соответственно ASP.NET MVC и ASP.NET Web API. Типы веб-приложений MVC и приложений API разделяют часть “модель” и “контроллер” паттерна MVC, в то время как веб-приложения MVC также реализуют “представление”, завершая паттерн MVC.

ASP.NET Core и .NET Core

Точно так же, как Entity Framework Core является полной переработкой Entity Framework 6, инфраструктура ASP.NET Core — это переработка популярной инфраструктуры ASP.NET Framework. Переписывание ASP.NET было нелегкой, но необходимой задачей, преследующей цель устраниить зависимости от System.Web. Избавление от указанной зависимости позволило запускать приложения ASP.NET под управлением операционных систем, отличающихся от Windows, и веб-серверов помимо Internet Information Services (IIS), включая размещаемые самостоятельно. В итоге у приложений ASP.NET Core появилась возможность использовать межплатформенный, легковесный и быстрый веб-сервер с открытым кодом под названием Kestrel, который предлагает унифицированный подход к разработке для всех платформ.

На заметку! Изначально продукт Kestrel был основан на LibUV, но после выпуска ASP.NET Core 2.1 он базируется на управляемых сокетах.

Подобно EF Core инфраструктура ASP.NET Core разрабатывается в виде проекта с полностью открытым кодом на GitHub (<https://github.com/aspnet>). Она также спроектирована как модульная система пакетов NuGet. Разработчики устанавливают только те функциональные средства, которые нужны для конкретного приложения, сводя к минимуму пространство памяти приложения, сокращая накладные расходы и снижая риски в плане безопасности. В число дополнительных улучшений входят упрощенный запуск, встроенное внедрение зависимостей, более чистая система конфигурирования и подключаемое промежуточное программное обеспечение (ПО).

Одна инфраструктура, много сценариев использования

В оставшихся главах этой части вы увидите, что в ASP.NET Core внесено множество изменений и усовершенствований. Помимо межплатформенных возможностей еще одним значительным изменением следует считать унификацию инфраструктур для создания веб-приложений. В рамках ASP.NET Core инфраструктуры ASP.NET MVC, ASP.NET Web API и Razor Pages объединены в единую инфраструктуру для разработки. Разработка веб-приложений и служб с применением полной инфраструктуры .NET Framework предоставляла несколько вариантов, включая Web Forms, MVC, Web API, Windows Communication Foundation (WCF) и WebMatrix. Все они имели положительные и отрицательные стороны; одни были тесно связаны между собой, а другие сильно отличались друг от друга. Наличие стольких доступных вариантов означало, что разработчики обязаны были знать каждый из них для выбора того, который подходит для имеющейся задачи, или просто отдавать предпочтение какому-то одному и положиться на удачу.

С помощью ASP.NET Core вы можете строить приложения, которые используют Razor Pages, паттерн MVC, службы REST и одностраничные приложения, где применяются библиотеки JavaScript вроде Angular либо React. Хотя визуализация пользовательского интерфейса зависит от выбора между MVC, Razor Pages или библиотеками JavaScript, лежащая в основе среда разработки остается той же самой. Двумя прежними вариантами, которые не были перенесены в ASP.NET Core, оказались Web Forms и WCF.

На заметку! Поскольку все обособленные инфраструктуры объединены вместе под одной крышей, прежние названия ASP.NET MVC и ASP.NET Web API официально были изъяты из употребления. Для простоты в этой книге веб-приложения ASP.NET Core, использующие паттерн "модель–представление–контроллер", упоминаются как MVC, а REST-службы ASP.NET — как Web API.

Функциональные средства ASP.NET Core из MVC/Web API

Многие проектные цели и функции, которые побудили разработчиков применять ASP.NET MVC и ASP.NET Web API, по-прежнему поддерживаются в ASP.NET Core (и были улучшены).

Ниже перечислены некоторые из них (но далеко не все):

- соглашения по конфигурации (*convention over configuration*; или соглашение над конфигурацией, если делать акцент на преимуществе соглашения перед конфигурацией);
- контроллеры и действия;
- привязка моделей;
- проверка достоверности моделей;
- маршрутизация;
- фильтры;
- компоновки и представления Razor.

Они рассматриваются в последующих разделах за исключением компоновок и представлений Razor, которые будут раскрыты в главе 31.

Соглашения по конфигурации

ASP.NET MVC и ASP.NET Web API сократили объем необходимой конфигурации за счет введения ряда соглашений. В случае соблюдения таких соглашений уменьшается объем ручного (или шаблонного) конфигурирования, но при этом разработчики обязаны знать соглашения, чтобы использовать их в своих интересах. Двумя главными видами соглашений являются соглашения об именовании и структура каталогов.

Соглашения об именовании

В ASP.NET Core существует множество соглашений об именовании, предназначенных для приложений MVC и API. Например, контроллеры обычно содержат суффикс `Controller` в своих именах (скажем, `HomeController`) и в добавок порождены от класса `Controller` (или `ControllerBase`). При доступе через маршрутизацию суффикс `Controller` отбрасывается. При поиске представлений контроллера отправной точкой поиска будет имя контроллера без суффикса. Такое соглашение об отбрасывании суффикса встречается повсюду в ASP.NET Core. В последующих главах вы встретите немало примеров.

Еще одно соглашение об именовании применяется относительно местоположения и выбора представлений. По умолчанию метод действия (в приложении MVC) будет визуализировать представление с таким же именем, как у метода. Шаблоны редактирования и отображения именуются в соответствии с классом, который они визуализируют в представлении. Описанное стандартное поведение может быть изменено, если того требует ваше приложение. Все это будет дополнительно исследоваться при построении приложения `AutoLot.Mvc`.

Структура каталогов

Существует несколько соглашений о папках, которые вы должны понимать, чтобы успешно создавать веб-приложения и службы ASP.NET Core.

Папка `Controllers`

По соглашению папка `Controllers` является тем местом, где реализации ASP.NET Core MVC и API (а также механизм маршрутизации) ожидают обнаружить контроллеры для вашего приложения.

Папка Views

В папке Views хранятся представления для приложения. Каждый контроллер получает внутри главной папки Views собственную папку с таким же именем, как у контроллера (исключая суффикс Controller). По умолчанию методы действий будут визуализировать представления из папки своего контроллера. Например, папка Views/Home содержит все представления для класса контроллера HomeController.

Папка Shared

Внутри папки Views имеется специальная папка по имени Shared, которая доступна всем контроллерам и их методам действий. Если представление не удалось найти в папке с именем контроллера, тогда поиск представления продолжается в папке Shared.

Папка wwwroot (нововведение в ASP.NET Core)

Улучшением по сравнению ASP.NET MVC стало создание особой папки по имени wwwroot для веб-приложений ASP.NET Core. В ASP.NET MVC файлы JavaScript, изображений, CSS и другое содержимое клиентской стороны смешивалось в остальных папках. В ASP.NET Core все файлы клиентской стороны содержатся в папке wwwroot. Такое отделение скомпилированных файлов от файлов клиентской стороны значительно проясняет структуру проекта при работе с ASP.NET Core.

Контроллеры и действия

Как и в ASP.NET MVC и ASP.NET Web API, контроллеры и методы действий являются основополагающими компонентами приложения ASP.NET Core MVC или API.

Класс Controller

Ранее уже упоминалось, что инфраструктура ASP.NET Core унифицировала ASP.NET MVC5 и ASP.NET Web API. Такая унификация также привела к объединению базовых классов Controller, ApiController и AsyncController из MVC5 и Web API 2.2 в один новый класс Controller, который имеет собственный базовый класс по имени ControllerBase. Контроллеры веб-приложений ASP.NET Core наследуются от класса Controller, тогда как контроллеры служб ASP.NET — от класса ControllerBase (рассматривается следующим). Класс Controller предлагает множество вспомогательных методов для веб-приложений, наиболее часто используемые из которых перечислены в табл. 29.1.

Таблица 29.1. Часто используемые вспомогательные методы, предоставляемые классом Controller

Вспомогательный метод	Описание
ViewDataTempDataViewBag ()	Снабжает представление данными через ViewDataDictionary, TempDataDictionary и динамический транспорт ViewBag
View()	Возвращает экземпляр класса ViewResult (производного от ActionResult) в качестве ответа HTTP. По умолчанию используется представление с таким же именем, как у метода действия, но есть возможность указать специфическое представление. Все варианты позволяют указывать модель представления, которая строго типизирована и передается View(). Представления раскрываются в главе 31

Окончание табл. 29.1

Вспомогательный метод	Описание
PartialView()	Возвращает экземпляр PartialViewResult в конвейер запросов. Частичные представления рассматриваются в главе 31
ViewComponent()	Возвращает экземпляр ViewComponentResult в конвейер запросов. Классы ViewComponents раскрываются в главе 31
Json()	Возвращает экземпляр JsonResult, который содержит объект, сериализованный в формате JSON, в качестве ответа
OnActionExecuting()	Выполняется перед выполнением метода действия
OnActionExecutionAsync()	Асинхронная версия OnActionExecuting()
OnActionExecuted()	Выполняется после выполнения метода действия

Класс ControllerBase

Класс ControllerBase обеспечивает основную функциональность для веб-приложений и служб ASP.NET Core, и вдобавок предлагает вспомогательные методы для возвращения кодов состояния HTTP. В табл. 29.2 описана основная функциональность класса ControllerBase, а в табл. 29.3 перечислены некоторые вспомогательные методы для возвращения кодов состояния HTTP.

Таблица 29.2. Часто используемые вспомогательные методы, предоставляемые классом ControllerBase

Вспомогательный метод	Описание
HttpContext()	Возвращает экземпляр HttpContext для действия, выполняющегося в текущий момент
Request()	Возвращает экземпляр HttpRequest для действия, выполняющегося в текущий момент
Response()	Возвращает экземпляр HttpResponseMessage для действия, выполняющегося в текущий момент
RouteData()	Возвращает экземпляр RouteData для действия, выполняющегося в текущий момент (маршрутизация рассматривается позже в этой главе)
ModelState()	Возвращает состояние модели для привязки и проверки достоверности (рассматриваются позже в этой главе)
Url()	Возвращает экземпляр реализации IUrlHelper, обеспечивающей доступ к построению URL для приложений и служб ASP.NET Core MVC
User()	Возвращает пользователя ClaimsPrincipal
Content()	Возвращает экземпляр ContentResult в ответ. Перегруженные версии позволяют добавлять определение типа содержимого и кодировки

Вспомогательный метод	Описание
File()	Возвращает экземпляр <code>FileContentResult</code> в ответ
Redirect()	Группа методов, которые выполняют перенаправление пользователя на другой URL, возвращая экземпляр <code>RedirectResult</code>
LocalRedirect()	Группа методов, которые выполняют перенаправление пользователя на другой URL, только если этот URL является локальным. Более безопасны, чем универсальные методы <code>Redirect()</code>
RedirectToAction()	Группа методов, которые выполняют перенаправление на другой метод действия, страницу Razor или именованный маршрут.
RedirectToPage()	Маршрутизация рассматривается позже в этой главе
RedirectToRoute()	
TryUpdateModel()	Явная привязка модели (расскрывается позже в главе)
TryValidateModel()	Явная проверка достоверности модели (рассматривается позже в главе)

Таблица 29.3. Часто используемые вспомогательные методы для возвращения кода состояния HTTP, предоставляемые классом `ControllerBase`

Вспомогательный метод	Результат действия	Код состояния HTTP
NoContent()	<code>NoContentResult</code>	204
Ok()	<code>OkResult</code>	200
NotFound()	<code>NotFoundResult</code>	404
BadRequest()	<code>BadRequestResult</code>	400
Created()	<code>CreatedResult</code>	201
CreatedAtAction()	<code>CreatedAtActionResult</code>	
CreatedAtRoute()	<code>CreateAtRouteResult</code>	
Accepted()	<code>AcceptedResult</code>	202
AcceptedAtAction()	<code>AcceptedAtActionResult</code>	
AcceptedAtRoute()	<code>AcceptedAtRouteResult</code>	

Действия

Действия — это методы в контроллере, которые возвращают экземпляр типа `IActionResult` (или `Task<IActionResult>` для асинхронных операций) либо класса, реализующего `IActionResult`, такого как `ActionResult` или `ViewResult`. Действия будут подробно рассматриваться в последующих главах.

Привязка моделей

Привязка моделей представляет собой процесс, в рамках которого инфраструктура ASP.NET Core использует пары “имя-значение”, отправленные HTTP-методом POST, для присваивания значений свойствам моделей. Для привязки к ссылочному типу пары “имя-значение” берутся из значений формы или тела запроса, ссылочные типы обязаны иметь открытый стандартный конструктор, а свойства, участвующие в привязке, должны быть открытыми и допускать запись. При присваивании значений

везде, где это применимо, используются неявные преобразования типов (вроде установки значения свойства `string` в значение `int`). Если преобразование типа терпит неудачу, тогда такое свойство помечается как имеющее ошибку. Прежде чем начать подробное обсуждение привязки, важно понять предназначение словаря `ModelState` и его роль в процессе привязки (а также проверки достоверности).

Словарь `ModelState`

Словарь `ModelState` содержит записи для всех привязываемых свойств и запись для самой модели. Если во время привязки возникает ошибка, то механизм привязки добавляет ее к записи словаря для свойства и устанавливает `ModelState.IsValid` в `false`. Если всем нужным свойствам были успешно присвоены значения, тогда механизм привязки устанавливает `ModelState.IsValid` в `true`.

На заметку! Проверка достоверности модели, которая тоже устанавливает записи словаря `ModelState`, происходит после привязки модели. Как неявная, так и явная привязка модели автоматически вызывает проверку достоверности для модели. Проверка достоверности рассматривается в следующем разделе.

Добавление специальных ошибок в словарь `ModelState`

В дополнение к свойствам и ошибкам, добавляемым механизмом привязки, в словарь `ModelState` можно добавлять специальные ошибки. Ошибки могут добавляться на уровне свойств или целой модели. Чтобы добавить специфическую ошибку для свойства (например, свойства `PetName` сущности `Car`), примените такой код:

```
ModelState.AddModelError("PetName", "Name is required");
```

Чтобы добавить ошибку для целой модели, указывайте в качестве имени свойства `string.Empty`:

```
ModelState.AddModelError(string.Empty,
    $"Unable to create record: {ex.Message}");
```

Неявная привязка моделей

Неявная привязка моделей происходит, когда привязываемая модель является параметром для метода действия. Для сложных типов она использует рефлексию и реурсию, чтобы сопоставить имена записываемых свойств модели с именами, которые содержатся в парах “имя-значение”, отправленных методу действия. При наличии совпадения имен средство привязки применяет значение из пары “имя-значение”, чтобы попробовать установить значение свойства. Если совпадение дают сразу несколько имен из пар “имя-значение”, тогда используется значение первого совпавшего имени. Если имя не найдено, то свойство устанавливается в стандартное значение для его типа. Вот как выглядит порядок поиска пар “имя-значение”:

- значения формы из HTTP-метода POST (включая отправки JavaScript AJAX);
- тело запроса (для контроллеров API);
- значения маршрута, предоставленные через маршрутизацию ASP.NET Core (для простых типов);
- значения строки запроса (для простых типов);
- загруженные файлы (для типов `IFormFile`).

Например, следующий метод будет пытаться установить все свойства в типе Car. Если процесс привязки завершается без ошибок, тогда свойство ModelState.IsValid возвращает true.

```
[HttpPost]
public ActionResult Create(Car entity)
{
    if (ModelState.IsValid)
    {
        // Сохранить данные.
    }
}
```

Явная привязка моделей

Явная привязка моделей запускается с помощью вызова метода TryUpdateModelAsync() с передачей ему экземпляра привязываемого типа и списка свойств, подлежащих привязке. Если привязка модели терпит неудачу, тогда метод возвращает false и устанавливает ошибки в ModelState аналогично неявной привязке. При использовании явной привязки моделей привязываемый тип не является параметром метода действия. Скажем, вы могли бы переписать предыдущий метод Create() с применением явной привязки:

```
[HttpPost]
public async Task<IActionResult> Create()
{
    var vm = new Car();
    if (await TryUpdateModelAsync(vm, "",
        c=>c.Color, c=>c.PetName, c=>c.MakeId, c=>c.TimeStamp))
    {
        // Делать что-то важное.
    }
}
```

Атрибут Bind

Атрибут Bind в HTTP-методах POST позволяет ограничить свойства, которые участвуют в привязке модели, или установить префикс для имени в парах “имя-значение”. Ограничение свойств, которые могут быть привязаны, снижает опасность атак избыточной отправкой (over-posting attack). Если атрибут Bind помещен на ссылочный параметр, то значения будут присваиваться через привязку модели только тем полям, которые перечислены в списке Include. Если атрибут Bind не используется, тогда привязку допускают все поля.

В следующем примере метода действия Create() все поля экземпляра Car доступны для привязки, поскольку атрибут Bind не применяется:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Car car)
{
    if (ModelState.IsValid)
    {
        // Добавить запись.
    }
    // Позволить пользователю повторить попытку.
}
```

Пусть в ваших бизнес-требованиях указано, что методу `Create()` разрешено обновлять только поля `PetName` и `Color`. Добавление атрибута `Bind` (как показано в примере ниже) ограничивает свойства, участвующие в привязке, и инструктирует средство привязки моделей о том, что остальные свойства должны игнорироваться.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(
    [Bind(nameof(Car.PetName), nameof(Car.Color))]Car car)
{
    if (ModelState.IsValid)
    {
        // Сохранить данные.
    }
    // Позволить пользователю повторить попытку.
}
```

Атрибут `Bind` можно также использовать для указания префикса имен свойств. Если имена в парах “имя-значение” имеют префикс, добавленный при их отправке методу действия, тогда атрибут `Bind` применяется для информирования средства привязки моделей о том, как сопоставлять эти имена со свойствами типа. Код в следующем примере устанавливает префикс для имен и позволяет привязывать все свойства:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(
    [Bind(Prefix="MakeList")]Car car)
{
    if (ModelState.IsValid)
    {
        // Сохранить данные.
    }
}
```

Управление источниками привязки моделей в ASP.NET Core

Источниками привязки можно управлять через набор атрибутов на параметрах действий. Допускается также создавать специальные средства привязки моделей, но эта тема выходит за рамки настоящей книги. В табл. 29.4 перечислены атрибуты, которые можно использовать для управления привязкой моделей.

Таблица 29.4. Управление источниками привязки моделей

Атрибут	Описание
<code>BindingRequired</code>	Если привязка не может произойти или свойству присваивается его стандартное значение, тогда будет добавлена ошибка состояния модели
<code>BindNever</code>	Сообщает средству привязки моделей о том, что параметр привязываться не должен
<code>FromHeader</code>	Используются для указания точного источника привязки, подлежащего применению (заголовок, строка запроса, параметры маршрута или значения формы)
<code>FromQuery</code>	
<code>FromRoute</code>	
<code>FromForm</code>	

Атрибут	Описание
FromServices	Привязывает тип с использованием внедрения зависимостей (рассматривается позже в главе)
FromBody	Привязывает данные из тела запроса. Форматер выбирается на основе содержимого запроса (скажем, JSON, XML и т.д.). Декорировать атрибутом FromBody можно не более одного параметра
ModelBinder	Применяется для переопределения стандартного средства привязки моделей (для специальной привязки моделей)

Проверка достоверности моделей

Проверка достоверности происходит немедленно после привязки модели (явной и неявной). В то время как привязка модели добавляет ошибки в словарь ModelState из-за возникновения проблем преобразования, проверка достоверности добавляет ошибки в ModelState на основе бизнес-правил. Примерами бизнес-правил могут быть обязательные поля, строки с максимально разрешенной длиной или даты с заданным допустимым диапазоном.

Правила проверки достоверности устанавливаются через атрибуты проверки достоверности, встроенные или специальные. В табл. 29.5 кратко описаны наиболее часто используемые встроенные атрибуты проверки достоверности. Обратите внимание, что некоторые из них также служат аннотациями данных для формирования сущностей EF Core.

Таблица 29.5. Часто используемые встроенные атрибуты проверки достоверности

Атрибут	Описание
CreditCard	Выполняет проверку по алгоритму Луна номера кредитной карты
Compare	Проверяет соответствие двух свойств модели
EmailAddress	Проверяет, имеет ли свойство допустимый формат адреса электронной почты
Phone	Проверяет, имеет ли свойство допустимый формат телефонного номера
Range	Проверяет, попадает ли значение свойства в указанный диапазон
RegularExpression	Проверяет, соответствует ли свойство указанному регулярному выражению
Required	Проверяет, имеет ли свойство значение
StringLength	Проверяет, не превышает ли длина свойства указанную максимальную длину
Url	Проверяет, имеет ли свойство допустимый формат URL
Remote	Проверяет введенные данные на клиентской стороне, вызывая метод действия на сервере

Можно также разработать специальные атрибуты проверки достоверности, но в книге данная тема не рассматривается.

Маршрутизация

Маршрутизация — это способ, которым ASP.NET Core сопоставляет HTTP-запросы с контроллерами и действиями (исполняемые *конечные точки*) в вашем приложении взамен старого процесса отображения URL на структуру файлов проекта, принятого в Web Forms. Она также предлагает механизм для создания URL изнутри приложения на основе таких конечных точек. Конечная точка в приложении MVC или Web API состоит из контроллера, действия (только MVC), метода HTTP и необязательных значений (называемых *значениями маршрута*).

На заметку! Маршруты также применяются к страницам Razor, SignalR, службам gRPC и т.д. В этой книге рассматриваются контроллеры стиля MVC и Web API.

Инфраструктура ASP.NET Core использует промежуточное ПО маршрутизации для сопоставления URL входящих запросов и для генерирования URL, отправляемых в ответах. Промежуточное ПО регистрируется в классе Startup, а конечные точки добавляются в классе Startup или через атрибуты маршрутов, как будет показано позже в главе.

Шаблоны URL и маркеры маршрутов

Конечные точки маршрутизации состоят из шаблонов URL, включающих в себя переменные-заполнители (называемые *маркерами*) и литералы, которые помещены в упорядоченную коллекцию, известную как *таблица маршрутов*. Каждая запись в ней определяет отличающийся шаблон URL, предназначенный для сопоставления. Заполнители могут быть специальными переменными или браться из заранее определенного списка. Зарезервированные маркеры маршрутов перечислены в табл. 29.6.

Таблица 29.6. Зарезервированные маркеры маршрутов для приложений MVC и API

Маркер	Описание
Area	Определяет область MVC для маршрута
Controller	Определяет имя контроллера (без суффикса Controller)
Action	Определяет имя действия в приложениях MVC

В дополнение к зарезервированным маркерам маршруты могут содержать специальные маркеры, которые отображаются (процессом привязки моделей) на параметры методов действий.

Маршрутизация и REST-службы ASP.NET Core

При определении маршрутов для служб ASP.NET метод действия не указывается. Вместо этого, как только контроллер обнаруживается, выполняемый метод действия базируется на методе HTTP запроса и назначениях методов HTTP методам действий. Детали будут приведены чуть позже.

Маршрутизация на основе соглашений

При маршрутизации на основе соглашений (или традиционной маршрутизации) таблица маршрутов строится в методе UseEndpoints() класса Startup. Метод MapControllerRoute() добавляет конечную точку в таблицу маршрутов, указывая

имя, шаблон URL и любые стандартные значения для переменных в шаблоне URL. В приведенном ниже примере кода заранее определенные заполнители {controller} и {action} ссылаются на контроллер и метод действия, содержащийся в данном контроллере. Заполнитель {id} является специальным и транслируется в параметр (по имени id) для метода действия. Добавление к маркеру маршрута знака вопроса указывает на его необязательность.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Запрашиваемый URL проверяется на соответствие с таблицей маршрутов. При наличии совпадения выполняется код, находящийся в этой конечной точке приложения. Примером URL, который мог бы обрабатываться таким маршрутом, является Car/Delete/5. В результате вызывается метод действия Delete() класса контроллера CarController с передачей значения 5 в параметре id.

В параметре default указано, каким образом заполнять пустые фрагменты в URL, которые содержат не все определенные компоненты. С учетом предыдущего кода, если в URL ничего не задано (например, <http://localhost:5001>), тогда механизм маршрутизации вызовет метод действия Index() класса HomeController без параметра id. Параметру default присуща поступательность, т.е. он допускает исключение справа налево. Однако пропускать части маршрута не разрешено. Ввод URL вида <http://localhost:5001/Delete/5> не пройдет сопоставление с шаблоном {controller}/{action}/{id}.

Механизм маршрутизации попытается отыскать первый маршрут на основе контроллера, действия, специальных маркеров и метода HTTP. Если механизм маршрутизации не может определить наилучший маршрут, тогда он генерирует исключение AmbiguousMatchException.

Обратите внимание, что шаблон маршрута не содержит протокол или имя хоста. Механизм маршрутизации автоматически добавляет в начало корректную информацию при создании маршрута и применяет метод HTTP, путь и параметры для определения соответствующей конечной точки приложения. Например, если ваш сайт запускается на <https://www.skimedic.com>, то протокол (HTTPS) и имя хоста (www.skimedic.com) автоматически добавляются к маршруту при его создании (скажем, <https://www.skimedic.com/Car/Delete/5>). Для входящего запроса механизм маршрутизации использует порцию Car/Delete/5 из URL.

Именованные маршруты

Имена маршрутов могут применяться в качестве сокращения для генерации URL изнутри приложения. Выше конечной точке было назначено имя default.

Маршрутизация с помощью атрибутов

При маршрутизации с помощью атрибутов маршруты определяются с использованием атрибутов C# в отношении контроллеров и их методов действий. Это может привести к более точной маршрутизации, но также увеличит объем конфигурации, поскольку для каждого контроллера и действия необходимо указать информацию маршрутизации.

Например, взгляните на приведенный ниже фрагмент кода. Четыре атрибута Route на методе действия Index() эквивалентны маршруту, который был определен ранее. Метод действия Index() является конечной точкой приложения для mysite.com, mysite.com/Home, mysite.com/Home/Index или mysite.com/Home/Index/5.

```
public class HomeController : Controller
{
    [Route("")]
    [Route("/Home")]
    [Route("/Home/Index")]
    [Route("/Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        ...
    }
}
```

Основное различие между маршрутизацией на основе соглашений и маршрутизацией с помощью атрибутов заключается в том, что первая охватывает приложение, тогда как вторая — контроллер с атрибутом Route. Если маршрутизация на основе соглашений не применяется, то каждому контроллеру понадобится определить свой маршрут, иначе доступ к нему будет невозможен. Скажем, если в таблице маршрутов не определен стандартный маршрут, тогда следующий код обнаружить не удастся, т.к. маршрутизация для контроллера не сконфигурирована:

```
public class CarController : Controller
{
    public IActionResult Delete(int id)
    {
        ...
    }
}
```

На заметку! Маршрутизацию на основе соглашений и маршрутизацию с помощью атрибутов можно использовать вместе. Если бы в методе UseEndpoints() был настроен стандартный маршрут контроллера (как в примере с маршрутизацией на основе соглашений), то предыдущий контроллер попал бы в таблицу маршрутов.

Когда маршруты добавляются на уровне контроллера, методы действий получают этот базовый маршрут. Например, следующий маршрут контроллера *охватывает* Delete() и *любые другие методы действий*:

```
[Route("[controller]/[action]/[id?]")]
public class CarController : Controller
{
    public IActionResult Delete(int id)
    {
        ...
    }
}
```

На заметку! При маршрутизации с помощью атрибутов встроенные маркеры помечаются квадратными скобками ([]), а не фигурными ({ }), как при маршрутизации на основе соглашений. Для специальных маркеров применяются все те же фигурные скобки.

Если методу действия необходимо перезапустить шаблон маршрута, тогда нужно предварить маршрут символом прямой косой черты (/). Скажем, если метод Delete() должен следовать шаблону URL вида `mysite.com/Delete/Car/5`, то вот как понадобится сконфигурировать действие:

```
[Route("[controller]/[action]/{id?}")]
public class CarController : Controller
{
    [Route("/{action}/{controller}/{id}")]
    public IActionResult Delete(int id)
    {
        ...
    }
}
```

В маршрутах также можно жестко кодировать значения маршрутов вместо замены маркеров. Показанный ниже код даст тот же самый результат, как и предыдущий:

```
[Route("[controller]/[action]/{id?}")]
public class CarController : Controller
{
    [Route("/Delete/Car/{id}")]
    public IActionResult Delete(int id)
    {
        ...
    }
}
```

Именованные маршруты

Маршрутам можно также назначать имена, что обеспечит сокращение для направления по определенному маршруту с указанием только его имени. Например, следующий атрибут маршрута имеет имя `GetOrderDetails`:

```
[HttpGet("{orderId}", Name = "GetOrderDetails")]
```

Маршрутизация и методы HTTP

Вы могли заметить, что ни в одном определении шаблона маршрута для методов не присутствует какой-нибудь метод HTTP. Причина в том, что механизм маршрутизации (в приложениях MVC и API) для выбора надлежащей конечной точки приложения использует шаблон маршрута и метод HTTP совместно.

Методы HTTP при маршрутизации в веб-приложениях (MVC)

Довольно часто при построении веб-приложений с применением паттерна MVC соответствовать определенному шаблону маршрута будут две конечные точки приложения. Средством различия в таких ситуациях является метод HTTP. Скажем, если `CarController` содержит два метода действий с именем `Delete()` и они оба соответствуют шаблону маршрута, то выбор метода для выполнения основывается на методе HTTP, который используется в запросе. Первый метод `Delete()` декорируется атрибутом `HttpGet` и будет выполняться, когда входящим запросом является GET. Второй метод `Delete()` декорируется атрибутом `HttpPost` и будет выполняться, когда входящим запросом является POST:

```
[Route("[controller]/[action]/{id?}")]
public class CarController : Controller
```

```
{
    [HttpGet]
    public IActionResult Delete(int id)
    {
        ...
    }

    [HttpPost]
    public IActionResult Delete(int id, Car recordToDelete)
    {
        ...
    }
}
```

Маршруты можно модифицировать также с применением атрибутов методов HTTP, а не атрибута Route. Например, ниже показан необязательный маркер маршрута id, добавленный в шаблон маршрута для обоих методов Delete():

```
[Route("[controller]/[action]")]
public class CarController : Controller
{
    [HttpGet("{id?}")]
    public IActionResult Delete(int? id)
    {
        ...
    }

    [HttpPost("{id}")]
    public IActionResult Delete(int id, Car recordToDelete)
    {
        ...
    }
}
```

Маршруты можно перезапускать с использованием методов HTTP; понадобится просто предварить шаблон маршрута символом прямой косой черты (/), как демонстрируется в следующем примере:

```
[HttpGet("/{controller}/{action}/{makeId}/{makeName}")]
public IActionResult ByMake(int makeId, string makeName)
{
    ViewBag.MakeName = makeName;
    return View(_repo.GetAllBy(makeId));
}
```

На заметку! Если метод действия не декорирован каким-либо атрибутом метода HTTP, то по умолчанию принимается метод GET. Тем не менее, в веб-приложениях MVC непомеченные методы действий могут также реагировать на запросы POST. По этой причине рекомендуется явно помечать все методы действий подходящим атрибутом метода HTTP.

Маршрутизация для служб API

Существенное различие между определениями маршрутов, которые применяются для приложений в стиле MVC, и определениями маршрутов, которые используются для служб REST, заключается в том, что в определениях маршрутов для служб не указываются методы действий. Методы действий выбираются на основе метода

HTTP запроса (и необязательно типа содержимого), но не по имени. Ниже приведен код контроллера API с четырьмя методами, которые все соответствуют одному и тому же шаблону маршрута. Обратите внимание на атрибуты методов HTTP:

```
[Route("api/{controller}")]
[ApiController]
public class CarController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetCarsById(int id)
    {
        ...
    }

    [HttpPost]
    public IActionResult CreateANewCar(Car entity)
    {
        ...
    }

    [HttpPut("{id}")]
    public IActionResult UpdateAnExistingCar(int id, Car entity)
    {
        ...
    }

    [HttpDelete("{id}")]
    public IActionResult DeleteACar(int id, Car entity)
    {
        ...
    }
}
```

Если метод действия не имеет атрибута метода HTTP, то он трактуется как конечная точка приложения для запросов GET. В случае если запрос соответствует маршруту, но метод действия с корректным атрибутом метода HTTP отсутствует, тогда сервер возвратит ошибку 404 (не найдено).

На заметку! Инфраструктура ASP.NET Web API позволяет не указывать метод HTTP для метода действия, если его имя начинается с Get, Put, Delete или Post. Следование такому соглашению обычно считалось плохой идеей и в ASP.NET Core оно было удалено. Если для метода действия не указан метод HTTP, то он будет вызываться с применением HTTP-метода GET.

Последним селектором конечных точек для контроллеров API является необязательный атрибут `Consumes`, который задает тип содержимого, принимаемый конечной точкой. В запросе должен использоваться соответствующий заголовок `content-type`, иначе будет возвращена ошибка 415 Unsupported Media Type (неподдерживаемый тип носителя). Следующие два примера конечных точек внутри одного и того же контроллера проводят различие между JSON и XML:

```
[HttpPost]
[Consumes("application/json")]
public IActionResult PostJson(IEnumerable<int> values) =>
    Ok(new { Consumes = "application/json", Values = values });
[HttpPost]
```

```
[Consumes("application/x-www-form-urlencoded")]
public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
    Ok(new { Consumes = "application/x-www-form-urlencoded", Values = values });
```

Перенаправление с использованием маршрутизации

Еще одно преимущество маршрутизации связано с тем, что вам больше не придется жестко кодировать URL для других страниц в своем сайте. Записи в таблице маршрутов применяются для сопоставления с входящими запросами, а также для построения URL. При построении URL схема, хост и порт добавляются на основе значений текущего запроса.

Фильтры

Фильтры в ASP.NET Core запускают код до или после специфической фазы конвейера обработки запросов. Существуют встроенные фильтры для авторизации и кеширования, а также возможность назначения специальных фильтров. В табл. 29.7 описаны типы фильтров, которые могут быть добавлены в конвейер, перечисленные в порядке их выполнения.

Таблица 29.7. Фильтры, доступные в ASP.NET Core

Фильтры	Описание
Фильтры авторизации	Запускаются первыми и определяют, авторизован ли пользователь для выполнения текущего запроса
Фильтры ресурсов	Запускаются сразу после фильтров авторизации и могут выполняться после того, как остаток конвейера завершен. Запускаются до привязки моделей
Фильтры действий	Запускаются немедленно перед выполнением действия и/или сразу после выполнения действия. Могут изменять значения, переданные в действие, и результат, возвращаемый из действия
Фильтры исключений	Используются для применения глобальных политик к необработанным исключениям, которые возникают перед записью в тело ответа
Фильтры результатов	Запускаются сразу после успешного выполнения результатов действий. Полезны для логики, которая окружает выполнение представления или форматера

Фильтры авторизации

Фильтры авторизации работают с системой ASP.NET Core Identity, чтобы предотвратить доступ к контроллерам или действиям, используя которые пользователь не имеет права. Создавать специальные фильтры авторизации не рекомендуется, поскольку встроенные классы `AuthorizeAttribute` и `AllowAnonymousAttribute` обычно обеспечивают достаточный охват в случае применения ASP.NET Core Identity.

Фильтры ресурсов

Код “перед” выполняется после фильтров авторизации и до любых других фильтров, а код “после” выполняется после всех остальных фильтров. Таким образом, фильтры ресурсов способны замкнуть накоротко целый конвейер запросов. Обычно фильтры ресурсов используются для кеширования. Если ответ находится в кеше, тогда фильтр может пропустить остаток конвейера.

Фильтры действий

Код “перед” выполняется немедленно перед выполнением метода действия, а код “после” выполняется сразу после выполнения метода действия. Фильтры действий могут замкнуть накоротко метод действия и любые фильтры, помещенные внутрь фильтров действий.

Фильтры исключений

Фильтры исключений реализуют сквозную обработку ошибок в приложении. У них нет событий, возникающих до или после, но они обрабатывают любые необработанные исключения, генерированные при создании контроллеров, привязке моделей, запуске фильтров действий либо выполнении методов действий.

Фильтры результатов

Фильтры результатов завершают выполнение экземпляра реализации `IActionResult` для метода действия. Распространенный сценарий применения фильтра результатов предусматривает добавление с его помощью информации заголовка в сообщение ответа HTTP.

Нововведения в ASP.NET Core

Помимо поддержки базовой функциональности ASP.NET MVC и ASP.NET Web API разработчики ASP.NET Core сумели добавить множество новых средств и улучшений в сравнении с предшествующими инфраструктурами. В дополнение к унификации инфраструктур и контроллеров появились следующие усовершенствования и инновации:

- встроенное внедрение зависимостей;
- система конфигурации, основанная на среде и готовая к взаимодействию с обличными технологиями;
- легковесный, высокопроизводительный и модульный конвейер запросов HTTP;
- вся инфраструктура основана на мелкозернистых пакетах NuGet;
- интеграция современных инфраструктур и рабочих потоков разработки для клиентской стороны;
- введение вспомогательных функций дескрипторов;
- введение компонентов представлений;
- огромные улучшения в плане производительности.

Встроенное внедрение зависимостей

Внедрение зависимостей (dependency injection — DI) представляет собой механизм для поддержки слабой связанныности между объектами. Вместо создания зависимых объектов напрямую или передачи специфических реализаций в классы и/или методы параметры определяются как интерфейсы. Таким образом, классам или методам и классам могут передаваться любые реализации интерфейсов, что разительно увеличивает гибкость приложения.

Поддержка DI — один из главных принципов, заложенных в переписанную версию ASP.NET Core. Все службы конфигурации и промежуточного ПО через внедрение зависимостей получает не только класс `Startup` (рассматриваемый позже в главе); ваши спе-

циальные классы могут (и должны) быть добавлены в контейнер DI с целью внедрения в другие части приложения. При конфигурировании элемента в контейнере ASP.NET Core DI доступны три варианта времени существования, кратко описанные в табл. 29.8.

Таблица 29.8. Варианты времени существования для служб

Вариант времени существования	Предоставляемая функциональность
Временный (transient)	Создается каждый раз, когда в нем возникает необходимость
С заданной областью (scoped)	Создается один раз для каждого запроса. Рекомендуется применять для объектов DbContext инфраструктуры Entity Framework Core
Одноэлементный (singleton)	Создается один раз при первом запросе и затем многократно используется во время существования объекта. Рекомендуется использовать вместо реализации класса в соответствие с паттерном "Одиночка" (Singleton)

Элементы в контейнере DI могут быть внедрены внутрь конструкторов и методов классов, а также в представления Razor.

На заметку! Если вы хотите использовать другой контейнер DI, то имейте в виду, что инфраструктура ASP.NET Core проектировалась с учетом такой гибкости. Чтобы узнать, как подключить другой контейнер DI, обратитесь в документацию по ссылке <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection>.

Осведомленность о среде

Осведомленность приложений ASP.NET Core об их среде выполнения включает переменные среды хоста и местоположения файлов через экземпляр реализации `IWebHostEnvironment`. В табл. 29.9 описаны свойства, доступные в этом интерфейсе.

Таблица 29.9. Свойства интерфейса `IWebHostEnvironment`

Свойство	Описание
<code>ApplicationName</code>	Получает или устанавливает имя приложения. По умолчанию используется имя сборки с точкой входа
<code>ContentRootPath</code>	Получает или устанавливает абсолютный путь к каталогу, в котором находятся файлы содержимого приложения
<code>ContentRootFileProvider</code>	Получает или устанавливает экземпляр реализации <code>IFileProvider</code> , указывающий на <code>ContentRootPath</code>
<code>EnvironmentName</code>	Получает или устанавливает имя среды. Устанавливается в значение переменной среды <code>ASPNETCORE_ENVIRONMENT</code>
<code>WebRootFileProvider</code>	Получает или устанавливает экземпляр реализации <code>IFileProvider</code> , указывающий на <code>WebRootPath</code>
<code>WebRootPath</code>	Получает или устанавливает абсолютный путь к каталогу, который содержит файлы содержимого приложения, обслуживаемые веб-сервером

Помимо доступа к важным файловым путям интерфейс `IWebHostEnvironment` применяется для выяснения среды времени выполнения.

Выяснение среды времени выполнения

Инфраструктура ASP.NET Core автоматически читает значение переменной среды по имени `ASPNETCORE_ENVIRONMENT`, чтобы установить среду времени выполнения. Если переменная `ASPNETCORE_ENVIRONMENT` не установлена, тогда ASP.NET Core устанавливает ее значение в `Production` (производственная среда). Установленное значение доступно через свойство `EnvironmentName` интерфейса `IWebHostEnvironment`.

Во время разработки приложений ASP.NET Core переменная `ASPNETCORE_ENVIRONMENT` обычно устанавливается с использованием файла настроек или командной строки. Последовательно идущие среды (подготовительная, производственная и т.д.), как правило, задействуют стандартные переменные среды операционной системы.

Вы можете применять для среды любое имя или одно из трех имен, которые предоставляются статическим классом `Environments`.

```
public static class Environments
{
    public static readonly string Development = "Development";
                                // среда разработки
    public static readonly string Staging = "Staging";
                                // подготовительная среда
    public static readonly string Production = "Production";
                                // производственная среда
}
```

Класс `HostEnvironmentEnvExtensions` предлагает расширяющие методы на `IHostEnvironment` для работы со свойством имени среды, которые описаны в табл. 29.10.

Таблица 29.10. Методы класса `HostEnvironmentEnvExtensions`

Метод	Описание
<code>IsProduction()</code>	Возвращает <code>true</code> , если переменная среды установлена в <code>Production</code> (нечувствительно к регистру символов)
<code>IsStaging()</code>	Возвращает <code>true</code> , если переменная среды установлена в <code>Staging</code> (нечувствительно к регистру символов)
<code>IsDevelopment()</code>	Возвращает <code>true</code> , если переменная среды установлена в <code>Development</code> (нечувствительно к регистру символов)
<code>IsEnvironment()</code>	Возвращает <code>true</code> , если переменная среды совпадает со строкой, переданной методу (нечувствительно к регистру символов)

Ниже перечислены некоторые примеры использования настройки среды:

- выяснение, какие конфигурационные файлы загружать;
- установка параметров отладки, ошибок и ведения журнала;
- загрузка файлов JavaScript и CSS, специфичных для среды.

Вы увидите все это в действии при построении приложений `AutoLot.Api` и `AutoLot.Mvc` в последующих двух главах.

Конфигурация приложений

В предшествующих версиях ASP.NET для конфигурирования служб и приложений применялся файл `web.config`, и разработчики получали доступ к конфигурационным настройкам через класс `System.Configuration`. Разумеется, помещение в файл `web.config` всех конфигурационных настроек для сайта, а не только специфичных для приложения, делало его (потенциально) запутанной смесью.

В ASP.NET Core была введена значительно более простая система конфигурации. По умолчанию она основывается на простых файлах JSON, которые хранят конфигурационные настройки в виде пар "имя-значение". Стандартный файл для конфигурации называется `appsettings.json`. Начальная версия файла `appsettings.json` (созданная шаблонами для веб-приложения ASP.NET Core и службы API) просто содержит конфигурационную информацию для регистрации в журнале, а также настройку для ограничения хостов:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Шаблон также создает файл `appsettings.Development.json`. Система конфигурации работает в сочетании с осведомленностью о среде времени выполнения, чтобы загружать дополнительные конфигурационные файлы на основе среды времени выполнения. Цель достигается инструктированием системы конфигурации о необходимости загрузки файла с именем `appsettings.{имя_среды}.json` после файла `appSettings.json`. В случае запуска приложения в среде разработки после файла начальных настроек загружается файл `appsettings.Development.json`. Если запуск происходит в подготовительной среде, тогда загружается файл `appsettings.Staging.json`. Важно отметить, что при загрузке более одного файла любые настройки, присутствующие в нескольких файлах, переопределяются настройками из последнего загруженного файла; они не являются аддитивными. Все конфигурационные настройки получаются через экземпляр реализации `IConfiguration`, доступный посредством системы внедрения зависимостей ASP.NET Core.

Извлечение настроек

После построения конфигурации к настройкам можно обращаться с использованием традиционного семейства методов `GetXXX()`, таких как `GetSection()`, `GetValue()` и т.д.:

```
Configuration.GetSection("Logging")
```

Также доступно сокращение для получения строк подключения:

```
Configuration.GetConnectionString("AutoLot")
```

Дополнительные возможности конфигурации будут повсеместно применяться в оставшемся материале книги.

Развертывание приложений ASP.NET Core

Приложения ASP.NET предшествующих версий могли развертываться только на серверах Windows с использованием IIS. Инфраструктуру ASP.NET Core можно разворачивать под управлением многочисленных операционных систем многими способами, в том числе и вне веб-сервера. Ниже перечислены высокуюровневые варианты:

- на сервере Windows (включая Azure) с применением IIS;
- на сервере Windows (включая службы приложений Azure) вне IIS;
- на сервере Linux с использованием Apache или NGINX;
- под управлением Windows или Linux в контейнере Docker.

Подобная гибкость позволяет организациям выбирать платформу развертывания, которая имеет наибольший смысл для организации, включая популярные модели развертывания на основе контейнеров (скажем, Docker), и не ограничиваться серверами Windows.

Легковесный и модульный конвейер запросов HTTP

Следуя принципам .NET Core, все в ASP.NET Core происходит по подписке. По умолчанию в приложение ничего не загружается. Такой подход позволяет приложениям быть насколько возможно легковесными, улучшая производительность и сводя к минимуму объем их кода и потенциальный риск.

Создание и конфигурирование решения

Теперь, когда у вас есть опыт работы с рядом основных концепций ASP.NET Core, самое время приступить к построению приложений ASP.NET Core. Проекты ASP.NET Core можно создавать с применением либо Visual Studio, либо командной строки. Оба варианта будут раскрыты в последующих двух разделах.

Использование Visual Studio

Преимущество Visual Studio связано с наличием графического пользовательского интерфейса, который поможет вам пройти через процесс создания решения и проектов, добавления пакетов NuGet и создания ссылок между проектами.

Создание решения и проектов

Начните с создания нового проекта в Visual Studio. Выберите в диалоговом окне Create a new project (Создание нового проекта) шаблон C# под названием ASP.NET Core Web Application (Веб-приложение ASP.NET Core). В диалоговом окне Configure your new project (Конфигурирование нового проекта) введите AutoLot.Api в качестве имени проекта и AutoLot для имени решения (рис. 29.1).

На следующем экране выберите шаблон ASP.NET Core Web API, а выше в раскрывающихся списках — .NET Core и ASP.NET Core 5.0. Оставьте флагшки внутри области Advanced (Дополнительно) в их стандартном состоянии (рис. 29.2).

Добавьте в решение еще один проект ASP.NET Core Web Application, выбрав шаблон ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (модель-представление-контроллер)). Удостоверьтесь в том, что в раскрывающихся списках вверху выбраны варианты .NET Core и ASP.NET Core 5.0; оставьте флагшки внутри области Advanced в их стандартном состоянии.

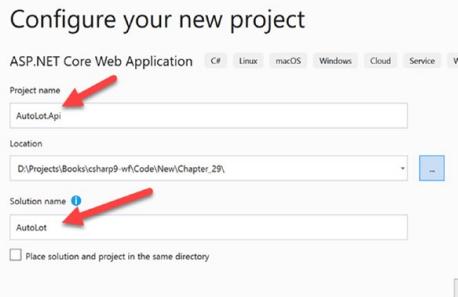


Рис. 29.1. Создание проекта AutoLot.Api и решения AutoLot

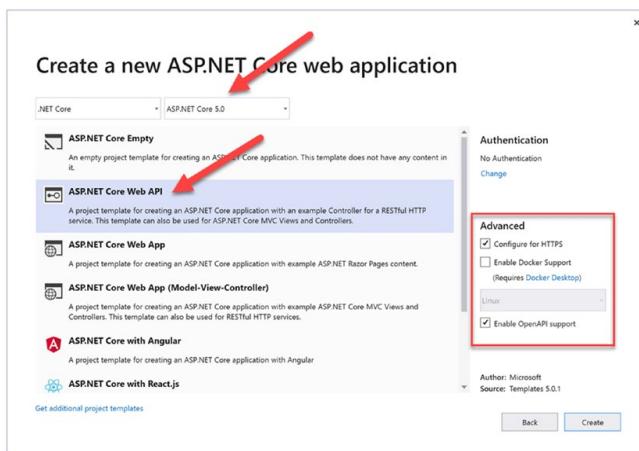


Рис. 29.2. Выбор шаблона ASP.NET Core Web API

Наконец, добавьте в решение проект C# Class Library (.NET Core) (Библиотека классов C# (.NET Core)) и назначьте ему имя AutoLot.Services. Отредактируйте файл проекта, чтобы установить TargetFramework в net5.0:

```
<PropertyGroup>
  <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
```

Добавление проектов AutoLot.Models и AutoLot.Dal

Решение требует завершенного уровня доступа к данным из главы 23. Вы можете либо скопировать файлы в каталог текущего решения, либо оставить их на месте. В любом случае вам нужно щелкнуть правой кнопкой мыши на имени решения в окне Solution Explorer, выбрать в контекстном меню пункт Add⇒Existing Project (Добавить⇒Существующий проект), перейти к файлу AutoLot.Models.csproj и выбрать его. Повторите такие же действия для проекта AutoLot.Dal.

На заметку! Хотя порядок добавления проектов в решение формально не имеет значения, среда Visual Studio сохранит ссылки между AutoLot.Models и AutoLot.Dal, если проект AutoLot.Models добавляется первым.

Добавление ссылок на проекты

Добавьте указанные ниже ссылки на проекты, щелкнув правой кнопкой на имени проекта в окне Solution Explorer и выбрав в контекстном меню пункт Add⇒Project Reference (Добавить⇒Ссылка на проект) для каждого проекта.

Проекты AutoLot.Api и AutoLot.Mvc ссылаются на:

- AutoLot.Models
- AutoLot.Dal
- AutoLot.Services

Проект AutoLot.Services ссылается на:

- AutoLot.Models
- AutoLot.Dal

Добавление пакетов NuGet

Для приложения необходимы дополнительные пакеты NuGet.

Добавьте перечисленные ниже пакеты в проект AutoLot.Api:

- AutoMapper
- System.Text.Json
- Swashbuckle.AspNetCore.Annotations
- Swashbuckle.AspNetCore.Swagger
- Swashbuckle.AspNetCore.SwaggerGen
- Swashbuckle.AspNetCore.SwaggerUI
- Microsoft.VisualStudio.Web.CodeGeneration.Design
- Microsoft.EntityFrameworkCore.SqlServer

На заметку! Благодаря шаблонам ASP.NET Core 5.0 API ссылка на Swashbuckle.AspNetCore уже присутствует. Указанные здесь пакеты Swashbuckle добавляют возможности за рамками базовой реализации.

Добавьте следующие пакеты в проект AutoLot.Mvc:

- AutoMapper
- System.Text.Json
- LigerShark.WebOptimizer.Core
- Microsoft.Web.LibraryManager.Build
- Microsoft.VisualStudio.Web.CodeGeneration.Design
- Microsoft.EntityFrameworkCore.SqlServer

Добавьте указанные ниже пакеты в проект AutoLot.Services:

- Microsoft.Extensions.Hosting.Abstractions
- Microsoft.Extensions.Options
- Serilog.AspNetCore
- Serilog.Enrichers.Environment
- Serilog.Settings.Configuration
- Serilog.Sinks.Console
- Serilog.Sinks.File
- Serilog.Sinks.MSSqlServer
- System.Text.Json

Использование командной строки

Как было показано ранее в книге, проекты и решения .NET Core можно создавать с применением командной строки. Откройте окно командной строки и перейдите в каталог, куда вы хотите поместить решение.

На заметку! В приводимых далее командах используется разделитель каталогов Windows. Если вы работаете не в среде Windows, тогда должным образом скорректируйте разделитель.

Создайте решение AutoLot и добавьте в него существующие проекты AutoLot.Models и AutoLot.Dal:

```
rem Создать решение
dotnet new sln -n AutoLot
rem Добавить в решение проекты
dotnet sln AutoLot.sln add ..\Chapter_23\AutoLot.Models
dotnet sln AutoLot.sln add ..\Chapter_23\AutoLot.Dal
```

Создайте проект AutoLot.Services, добавьте его в решение, добавьте пакеты NuGet и добавьте ссылки на проекты:

```
rem Создать библиотеку классов для служб приложения и добавить ее в решение
dotnet new classlib -lang c# -n AutoLot.Services
-o .\AutoLot.Services -f net5.0
dotnet sln AutoLot.sln add AutoLot.Services

rem Добавить пакеты
dotnet add AutoLot.Services
    package Microsoft.Extensions.Hosting.Abstractions
dotnet add AutoLot.Services package Microsoft.Extensions.Options
dotnet add AutoLot.Services package Serilog.AspNetCore
dotnet add AutoLot.Services package Serilog.Enrichers.Environment
dotnet add AutoLot.Services package Serilog.Settings.Configuration
dotnet add AutoLot.Services package Serilog.Sinks.Console
dotnet add AutoLot.Services package Serilog.Sinks.File
dotnet add AutoLot.Services package Serilog.Sinks.MSSqlServer
dotnet add AutoLot.Services package System.Text.Json

rem Добавить ссылки на проекты
dotnet add AutoLot.Services reference ..\Chapter_23\AutoLot.Models
dotnet add AutoLot.Services reference ..\Chapter_23\AutoLot.Dal
```

Создайте проект AutoLot.Api, добавьте его в решение, добавьте пакеты NuGet и добавьте ссылки на проекты:

```
dotnet new webapi -lang c# -n AutoLot.Api -au none
-o .\AutoLot.Api -f net5.0
dotnet sln AutoLot.sln add AutoLot.Api

rem Добавить пакеты
dotnet add AutoLot.Api package AutoMapper
dotnet add AutoLot.Api package Swashbuckle.AspNetCore
dotnet add AutoLot.Api package Swashbuckle.AspNetCore.Annotations
dotnet add AutoLot.Api package Swashbuckle.AspNetCore.Swagger
dotnet add AutoLot.Api package Swashbuckle.AspNetCore.SwaggerGen
dotnet add AutoLot.Api package Swashbuckle.AspNetCore.SwaggerUI
dotnet add AutoLot.Api
    package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add AutoLot.Api package Microsoft.EntityFrameworkCore.SqlServer
dotnet add AutoLot.Api package System.Text.Json

rem Добавить ссылки на проекты
dotnet add AutoLot.Api reference ..\Chapter_23\AutoLot.Dal
dotnet add AutoLot.Api reference ..\Chapter_23\AutoLot.Models
dotnet add AutoLot.Api reference AutoLot.Services
```

Создайте проект AutoLot.Mvc, добавьте его в решение, добавьте пакеты NuGet и добавьте ссылки на проекты:

```
dotnet new mvc -lang c# -n AutoLot.Mvc -au none -o .\AutoLot.Mvc -f net5.0
dotnet sln AutoLot.sln add AutoLot.Mvc

rem Добавить ссылки на проекты
dotnet add AutoLot.Mvc reference ..\Chapter_23\AutoLot.Models
dotnet add AutoLot.Mvc reference ..\Chapter_23\AutoLot.Dal
dotnet add AutoLot.Mvc reference AutoLot.Services

rem Добавить пакеты
dotnet add AutoLot.Mvc package AutoMapper
dotnet add AutoLot.Mvc package System.Text.Json
dotnet add AutoLot.Mvc package LigerShark.WebOptimizer.Core
dotnet add AutoLot.Mvc package Microsoft.Web.LibraryManager.Build
dotnet add AutoLot.Mvc package Microsoft.EntityFrameworkCore.SqlServer
dotnet add AutoLot.Mvc
    package Microsoft.VisualStudio.Web.CodeGeneration.Design
```

На этом настройка с применением командной строки завершена. Она намного эффективнее, если вы не нуждаетесь в графическом пользовательском интерфейсе Visual Studio.

Запуск приложений ASP.NET Core

Веб-приложения предшествующих версий ASP.NET всегда запускались с использованием IIS (или IIS Express). В ASP.NET Core приложения обычно запускаются с применением веб-сервера Kestrel, но существует вариант использования IIS, Apache, Nginx и т.д. через обратный прокси-сервер между Kestrel и другим веб-сервером. В результате происходит не только отход в сторону от строгого применения IIS, чтобы сменить модель развертывания, но и изменение возможностей разработки.

Во время разработки приложения теперь можно запускать следующим образом:

- из Visual Studio с использованием IIS Express;
- из Visual Studio с применением Kestrel;
- из командной строки .NET CLI с использованием Kestrel;
- из Visual Studio Code (VS Code) через меню Run (Запуск) с применением Kestrel;
- из VS Code через окно терминала с использованием .NET CLI и Kestrel.

Конфигурирование настроек запуска

Файл `launchsettings.json` (расположенный в узле Properties (Свойства) окна Solution Explorer) конфигурирует способ запуска приложения во время разработки под управлением Kestrel и IIS Express. Ниже приведено его содержимое в справочных целях (ваши порты IIS Express будут другими):

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:42788",
      "sslPort": 44375
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "AutoLot.Api": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Использование Visual Studio

В разделе `iisSettings` определены настройки запуска приложения с применением IIS Express в качестве веб-сервера. Наиболее важными настройками, на которые следует обратить внимание, являются настройка `applicationUrl`, определяющая порт, и блок `environmentVariables`, где определяется среда выполнения.

При запуске приложения в режиме отладки эта настройка заменяет собой любую настройку среды машины. Второй профиль (`AutoLot.Mvc` или `AutoLot.Api` в зависимости от того, какой проект используется) определяет настройки для ситуации, когда приложение запускается с применением Kestrel в качестве веб-сервера. Профиль определяет `applicationUrl` и порты плюс среду.

Меню Run в Visual Studio позволяет выбрать либо IIS Express, либо Kestrel, как показано на рис. 29.3. После выбора профиля проект можно запустить, нажав `<F5>` (режим отладки), нажав `<Ctrl+F5>` (эквивалентно выбору пункта Start Without Debugging (Запустить без отладки) в меню Debug (Отладка)) или щелкнув на кнопке запуска с зеленой стрелкой (эквивалентно выбору пункта Start Debugging (Запустить с отладкой) в меню Debug).

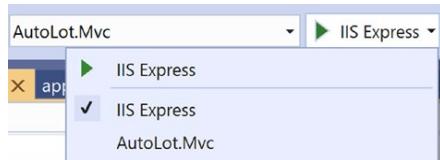


Рис. 29.3. Профили отладки, доступные в Visual Studio

На заметку! В случае запуска приложения из Visual Studio средства редактирования и продолжения больше не поддерживаются.

Использование командной строки или окна терминала Visual Studio Code

Чтобы запустить приложение из командной строки или окна терминала VS Code, перейдите в каталог, где находится файл `.csproj` для вашего приложения. Введите следующую команду для запуска приложения под управлением веб-сервера Kestrel:

```
dotnet run
```

Для завершения процесса нажмите `<Ctrl+C>`.

Изменение кода во время отладки

При запуске из командной строки код можно изменять, но изменения никак не будут отражаться в выполняющемся приложении. Чтобы изменения отражались в выполняющем приложении, введите такую команду:

```
dotnet watch run
```

Обновленная команда вместе с вашим приложением запускает средство наблюдения за файлами. Когда в файлах любого проекта (или проекта, на который имеется ссылка) обнаруживаются изменения, приложение автоматически останавливается и затем снова запускается. Нововведением в версии ASP.NET Core 5 является перезагрузка любых подключенных окон браузера. Хотя в итоге средства редактирования и продолжения в точности не воспроизводятся, это немалое удобство при разработке.

Использование Visual Studio Code

Чтобы запустить проекты в VS Code, откройте каталог, где находится решение. После нажатия `<F5>` (или щелчка на кнопке запуска) среда VS Code предложит вы-

брать проект для запуска (`AutoLot.Api` или `AutoLot.Mvc`), создаст конфигурацию запуска и поместит ее в файл по имени `launch.json`. Кроме того, среда VS Code использует файл `launchsettings.json` для чтения конфигурации портов.

Изменение кода во время отладки

В случае запуска приложения из VS Code код можно изменять, но изменения никак не будут отражаться в выполняющемся приложении. Чтобы изменения отражались в выполняющем приложении, введите в окне терминала команду `dotnet watch run`.

Отладка приложений ASP.NET Core

При запуске приложения из Visual Studio или VS Code отладка работает вполне ожидаемым образом. Но при запуске из командной строки вам необходимо присоединиться к выполняющемуся процессу, прежде чем вы сможете отлаживать свое приложение. В Visual Studio и VS Code это делается легко.

Присоединение с помощью Visual Studio

После запуска приложения (посредством команды `dotnet run` или `dotnet watch run`) выберите пункт меню `Debug⇒Attach to Process` (Отладка⇒Присоединиться к процессу) в Visual Studio. В открывшемся диалоговом окне `Attach to Process` (Присоединение к процессу) отыщите процесс по имени вашего приложения (рис. 29.4).

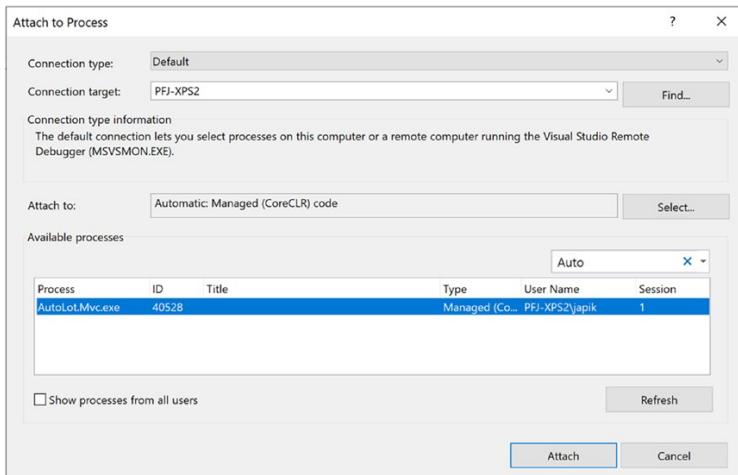


Рис. 29.4. Присоединение к выполняющемуся приложению для его отладки в Visual Studio

После присоединения к выполняющемуся процессу вы можете устанавливать в Visual Studio точки останова, и отладка будет работать так, как ожидалось. Редактировать и продолжать выполнение не удастся; чтобы изменения отразились в функционирующем приложении, придется отсоединиться от процесса.

Присоединение с помощью Visual Studio Code

После запуска приложения (командой `dotnet run` или `dotnet watch run`) щелкните на кнопке запуска с зеленой стрелкой и выберите `.NET Core Attach` (Присоединение .NET Core) вместо `.NET Core Launch (web)` (Запуск .NET Core (веб)), как показано на рис. 29.5.

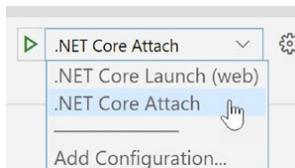


Рис. 29.5. Присоединение к выполняющемуся приложению для его отладки в VS Code

Когда вы щелкнете на кнопке запуска, вам будет предложено выбрать процесс для присоединения к нему. Выберите свое приложение. Затем можете устанавливать точки останова обычным образом.

Преимущество использования среды VS Code заключается в том, что после ее присоединения (и применения команды `dotnet watch run`) вы можете обновлять свой код во время выполнения (без необходимости в отсоединении) и вносимые изменения будут отражаться в функционирующем приложении.

Обновление портов AutoLot.Api

Вы могли заметить, что приложения AutoLot.Api и AutoLot.Mvc имеют различные порты, указанные для их профилей IIS Express, но для обоих приложений порты Kestrel установлены в 5000 (HTTP) и 5001 (HTTPS), что вызовет проблемы, когда вы попытаетесь запустить приложения вместе. Измените порты для AutoLot.Api на 5020 (HTTP) и 5021 (HTTPS), например:

```
"AutoLot.Api": {
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "api/values",
  "applicationUrl": "https://localhost:5021;http://localhost:5020",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Создание и конфигурирование экземпляраWebHost

В отличие от классических приложений ASP.NET MVC или ASP.NET Web API приложения ASP.NET Core — это просто консольные приложения .NET Core, которые создают и конфигурируют экземпляр `WebHost`. Создание экземпляра `WebHost` и его последующее конфигурирование обеспечивают настройку приложения на прослушивание (и реагирование) на запросы HTTP. Экземпляр `WebHost` создается в методе `Main()` внутри файла `Program.cs`. Затем экземпляр `WebHost` конфигурируется для вашего приложения в файле `Startup.cs`.

Файл Program.cs

Откройте файл класса `Program.cs` в приложении `AutoLot.Api` и просмотрите его содержимое, которое для справки приведено ниже:

```
namespace AutoLot.Api
{
  public class Program
  {
    public static void Main(string[] args)
    {
      CreateHostBuilder(args).Build().Run();
    }
}
```

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
}

```

Метод `CreateDefaultBuilder()` ужимает наиболее типовую настройку приложения в один вызов. Он конфигурирует приложение (используя переменные среды и JSON-файлы `appsettings`), настраивает стандартный поставщик ведения журнала и устанавливает контейнер DI. Такая настройка обеспечивается шаблонами ASP.NET Core для приложений API и MVC.

Следующий метод, `ConfigureWebHostDefaults()`, тоже является мета-методом, который добавляет поддержку для Kestrel, IIS и дополнительные настройки. Финальный шаг представляет собой установку класса конфигурации, специфичной для приложения, который в данном примере (и по соглашению) называется `Startup`. Наконец, вызывается метод `Run()` для активизации экземпляра `WebHost`.

Помимо экземпляра `WebHost` в предыдущем коде создается экземпляр реализации `IConfiguration`, который добавляется внутрь контейнера DI.

Файл `Startup.cs`

Класс `Startup` конфигурирует то, как приложение будет обрабатывать запросы и ответы HTTP, настраивает необходимые службы и добавляет службы в контейнер DI. Имя класса может быть любым, если оно соответствует строке `UseStartup<T>()` в конфигурации метода `CreateHostBuilder()`, но по соглашению класс именуется как `Startup`.

Доступные службы для класса `Startup`

Процессу запуска требуется доступ к инфраструктуре, а также к службам и настройкам среды, которые внедряются в класс инфраструктурой. Классу `Startup` доступно пять служб для конфигурирования приложения, которые кратко описаны в табл. 29.11.

Таблица 29.11. Службы, доступные классу `Startup`

Служба	Описание
<code>IApplicationBuilder</code>	Определяет класс, который предоставляет механизмы для конфигурирования конвейера запросов приложения
<code>IWebHostEnvironment</code>	Предоставляет информацию о среде веб-хостинга, в которой выполняется приложение
<code>ILoggerFactory</code>	Применяется для конфигурирования системы ведения журнала и создания средств ведения журнала из зарегистрированных поставщиков
<code>IServiceCollection</code>	Указывает контракт для набора дескрипторов служб. Является частью инфраструктуры внедрения зависимостей
<code>IConfiguration</code>	Экземпляр конфигурации приложения, созданный в методе <code>Main()</code> класса <code>Program</code>

Конструктор принимает экземпляр реализации `IConfiguration` и необязательный экземпляр реализации `IWebHostEnvironment/IHostEnvironment`. Метод `ConfigureServices()` запускается до того, как метод `Configure()` получает экземпляр реализации `IServiceCollection`. Метод `Configure()` должен принимать экземпляр реализации `IApplicationBuilder`, но может принимать экземпляры реализаций `IWebHostEnvironment/IHostEnvironment`, `ILoggerFactory` и любых интерфейсов, которые были добавлены внутрь контейнера DI в методе `ConfigureServices()`. Все перечисленные компоненты обсуждаются в последующих разделах.

Конструктор

Конструктор принимает экземпляр реализации интерфейса `IConfiguration`, который был создан методом `Host.CreateDefaultBuilder` в файле класса `Program.cs`, и присваивает его свойству `Configuration` для использования где-то в другом месте внутри класса. Конструктор также может принимать экземпляр реализации `IWebHostEnvironment` и/или `ILoggerFactory`, хотя он не добавляется в стандартном шаблоне.

Добавьте в конструктор параметр для `IWebHostEnvironment` и присвойте его локальной переменной уровня класса. Это понадобится в методе `ConfigureServices()`. Проделайте такую же работу для приложений `AutoLot.Api` и `AutoLot.Mvc`.

```
private readonly IWebHostEnvironment _env;
public Startup(
    IConfiguration configuration, IWebHostEnvironment env)
{
    _env = env;
    Configuration = configuration;
}
```

Метод `ConfigureServices()`

Метод `ConfigureServices()` применяется для конфигурирования любых служб, необходимых приложению, и вставки их в контейнер DI. Сюда входят службы, требуемые для поддержки приложений MVC и служб API.

AutoLot.Api

Метод `ConfigureServices()` для API-интерфейса `AutoLot` по умолчанию конфигурируется с только одной службой, которая добавляет поддержку контроллеров. Благодаря этому мета-методу добавляется множество дополнительных служб, в том числе маршрутизация, авторизация, привязка моделей и все элементы, не относящиеся к пользовательскому интерфейсу, которые уже обсуждались в настоящей главе.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}
```

Метод `AddControllers()` может быть расширен, например, для настройки обработки JSON. По умолчанию для ASP.NET Core используется “верблюжий” стиль при обработке JSON (первая буква в нижнем регистре, а каждое последующее слово начинается с буквы верхнего регистра, скажем, `carRepo`). Это соответствует большинству инфраструктур производства не Microsoft, которые применяются для разработки веб-

приложений. Однако в предшествующих версиях ASP.NET использовался стиль Pascal (например, CarRepo). Переход на “верблюжий” стиль был критическим изменением для многих приложений, которые ожидали стиля Pascal. Чтобы вернуть стиль Pascal при обработке JSON приложению (и улучшить форматирование разметки JSON), модифицируйте метод ConfigureServices() следующим образом:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers()
        .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.PropertyNamingPolicy = null;
        options.JsonSerializerOptions.WriteIndented = true;
    });
}
```

Добавьте в файл Startup.cs перечисленные ниже операторы using:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

Службам API необходим доступ к ApplicationContext и хранилищам внутри уровня доступа к данным. Существует встроенная поддержка для добавления EF Core в приложения ASP.NET Core. Добавьте следующий код в метод ConfigureServices() класса Startup:

```
var connectionString = Configuration.GetConnectionString("AutoLot");
services.AddDbContextPool<ApplicationContext>(
    options => options.UseSqlServer(connectionString,
        sqlOptions => sqlOptions.EnableRetryOnFailure()));
```

Первая строка кода получает строку подключения из файла настроек (более подробно рассматривается позже). Следующая строка добавляет в контейнер DI пул экземпляров ApplicationContext. Во многом подобно пулу подключений пул ApplicationContext может улучшить показатели производительности за счет наличия заранее установленных экземпляров, ожидающих потребления. Когда нужен контекст, он загружается из пула. По окончании его использования он очищается от любых следов применения и возвращается в пул.

Теперь необходимо добавить хранилища в контейнер DI. Вставьте в метод ConfigureServices() приведенный далее код после кода для конфигурирования ApplicationContext:

```
services.AddScoped<ICarRepo, CarRepo>();
services.AddScoped<ICreditRiskRepo, CreditRiskRepo>();
services.AddScoped<ICustomerRepo, CustomerRepo>();
services.AddScoped<IMakeRepo, MakeRepo>();
services.AddScoped<IOrderRepo, OrderRepo>();
```

Добавление строки подключения к настройкам приложения

Модифицируйте файл appsettings.development.json, как показано ниже, добавив строку подключения к базе данных. Обязательно включите запятую, отделяющую разделы, и приведите строку подключения в соответствие со своей средой.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionStrings": {
    "AutoLot": "Server=.;5433;Database=AutoLotFinal;
    User ID=sa;Password=P@ssw0rd;"
  }
}
```

Как обсуждалось ранее, каждый конфигурационный файл именуется согласно среде, что позволяет разносить значения, специфичные к среде, по разным файлам. Добавьте в проект новый файл по имени appsettings.production.json и обновите его следующим образом:

```
{
  "ConnectionStrings": {
    "AutoLot": "ITSASECRET"
  }
}
```

Это предохраняет реальную строку подключения от системы управления версиями и делает возможным замену маркера (ITSASECRET) в течение процесса разработки.

AutoLot.Mvc

Метод ConfigureServices() для веб-приложений MVC добавляет базовые службы для приложений API и поддержку визуализации представлений. Вместо вызова AddControllers() в приложениях MVC вызывается AddControllersWithViews():

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
}
```

Добавьте в файл Startup.cs показанные ниже операторы using:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

Веб-приложение также должно использовать уровень доступа к данным. Добавьте в метод ConfigureServices() класса Startup следующий код:

```
var connectionString = Configuration.GetConnectionString("AutoLot");
services.AddDbContextPool<ApplicationContext>(
  options => options.UseSqlServer(connectionString,
    sqlOptions => sqlOptions.EnableRetryOnFailure()));
services.AddScoped<ICarRepo, CarRepo>();
services.AddScoped<ICreditRiskRepo, CreditRiskRepo>();
services.AddScoped<ICustomerRepo, CustomerRepo>();
services.AddScoped<IMakeRepo, MakeRepo>();
services.AddScoped<IOrderRepo, OrderRepo>();
```

На заметку! Веб-приложение MVC будет работать как с уровнем доступа к данным, так и с API-интерфейсом для взаимодействия с данными, чтобы продемонстрировать оба механизма.

Добавление строки подключения к настройкам приложения

Модифицируйте файл appsettings.development.json, как показано ниже:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "ConnectionStrings": {
    "AutoLot": "Server=.;5433;Database=AutoLotFinal;
    User ID=sa;Password=P@ssw0rd;"
  }
}
```

Метод Configure()

Метод Configure() применяется для настройки приложения на реагирование на запросы HTTP. Данный метод выполняется *после* метода ConfigureServices(), т.е. все, что добавлено в контейнер DI, также может быть внедрено в Configure().

Существуют различия в том, как приложения API и MVC конфигурируются для обработки запросов и ответов HTTP в конвейере.

AutoLot.Api

Внутри стандартного шаблона выполняется проверка среды, и если она установлена в Development (среда разработки), тогда в конвейер обработки добавляется промежуточное ПО UseDeveloperExceptionPage(), предоставляющее отладочную информацию, которую вы вряд ли захотите отображать в производственной среде.

Далее производится вызов UseHttpsRedirection() для перенаправления всего трафика на HTTPS (вместо HTTP). Затем добавляются вызовы app.UseRouting(), app.UseAuthorization() и app.UseEndpoints(). Вот полный код метода:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  if (env.IsDevelopment())
  {
    // Если среда разработки, тогда отображать отладочную информацию.
    app.UseDeveloperExceptionPage();
    // Первонаучальный код.
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
      "AutoLot.Api v1"));
  }
  // Перенаправить трафик HTTP на HTTPS.
  app.UseHttpsRedirection();
  // Включить маршрутизацию.
  app.UseRouting();
```

514 Часть IX. ASP.NET Core

```
// Включить проверки авторизации.
app.UseAuthorization();

// Включить маршрутизацию с использованием конечных точек.
// Использовать для контроллеров маршрутизацию с помощью атрибутов.
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
}
```

Кроме того, когда приложение запускается в среде разработки, необходимо инициализировать базу данных. Добавьте в метод `Configure()` параметр типа `ApplicationContext` и вызовите метод `InitializeData()` из `AutoLot.Dal`.

Ниже показан модифицированный код:

```
public void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env,
    ApplicationContext context)
{
    if (env.IsDevelopment())
    {
        // Если среда разработки, тогда отображать отладочную информацию.
        app.UseDeveloperExceptionPage();

        // Инициализировать базу данных.
        if (Configuration.GetValue<bool>("RebuildDataBase"))
        {
            SampleDataInitializer.InitializeData(context);
        }
    }
    ...
}
```

Обновите файл `appsettings.development.json` с учетом свойства `Rebuild DataBase` (пока что установив его в `false`):

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "RebuildDataBase: false,
    "ConnectionStrings": {
        "AutoLot": "Server=db;Database=AutoLotPresentation;
                    User ID=sa;Password=P@ssw0rd;"
    }
}
```

AutoLot.Mvc

Метод `Configure()` для веб-приложений немного сложнее, чем его аналог для API.

Ниже приведен полный код метода с последующим обсуждением:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Метод `Configure()` также проверяет среду, и если она установлена в `Development` (среда разработки), тогда в конвейер обработки добавляется промежуточное ПО `UseDeveloperExceptionPage()`. Для любой другой среды в конвейер обработки добавляется универсальное промежуточное ПО `UseExceptionHandler()` и поддержка протокола строгой транспортной безопасности HTTP (HTTP Strict Transport Security — HSTS). Как и в аналоге для API, добавляется вызов `app.UseHttpsRedirection()`. Следующим шагом является добавление поддержки статических файлов с помощью вызова `app.UseStaticFiles()`. Поддержка статических файлов включается как мера по усилению безопасности. Если ваше приложение в ней не нуждается (подобно API-интерфейсам), тогда не добавляйте такую поддержку. Затем добавляется промежуточное ПО для маршрутизации, авторизации и конечных точек.

Добавьте в метод параметр типа `ApplicationContext` и вызовите `InitializeData()` из `AutoLot.Dal`. Вот модифицированный код:

```
public void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env,
    ApplicationContext context)
{
    if (env.IsDevelopment())
    {
        // Если среда разработки, тогда отображать отладочную информацию.
        app.UseDeveloperExceptionPage();
        // Инициализировать базу данных.
        if (Configuration.GetValue<bool>("RebuildDataBase"))
        {
            SampleDataInitializer.InitializeData(context);
        }
    }
    ...
}
```

Обновите файл appsettings.development.json с учетом свойства RebuildDataBase (пока что установив его в false):

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "RebuildDataBase": false,
  "ConnectionStrings": {
    "AutoLot": "Server=db;Database=AutoLotPresentation;
    User ID=sa;Password=P@ssw0rd;"
  }
}
```

Стандартный шаблон настраивает в методе UseEndpoints() маршрутизацию на основе соглашений. Ее понадобится отключить и повсюду в приложении применять маршрутизацию с помощью атрибутов. Закомментируйте (или удалите) вызов MapControllerRoute() и замените его вызовом MapControllers():

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllers();
});
```

Далее добавьте атрибуты маршрутов к HomeController в приложении AutoLot.Mvc. Первым делом добавьте шаблон контроллер/действие к самому контроллеру:

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
  ...
}
```

Затем добавьте три маршрута к методу Index(), так что он будет стандартным действием, когда не указано действие либо когда не указан контроллер или действие. Кроме того, снабдите метод атрибутомHttpGet, чтобы явно объявить его действием GET:

```
[Route("/")]
[Route("/{controller}")]
[Route("/{controller}/{action}")]
[HttpGet]
public IActionResult Index()
{
  return View();
}
```

Ведение журнала

Базовая инфраструктура ведения журнала добавляется в контейнер DI как часть процесса запуска и конфигурирования. Инфраструктура ведения журнала использует довольно простой интерфейс ILogger<T>. Основополагающим компонентом ведения журнала является класс LoggerExtensions, определения методов которого показаны ниже:

```
public static class LoggerExtensions
{
    public static void LogDebug(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogDebug(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogDebug(this ILogger logger, Exception exception,
        string message, params object[] args)
    public static void LogDebug(this ILogger logger,
        string message, params object[] args)

    public static void LogTrace(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogTrace(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogTrace(this ILogger logger, Exception exception,
        string message, params object[] args)
    public static void LogTrace(this ILogger logger,
        string message, params object[] args)

    public static void LogInformation(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogInformation(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogInformation(this ILogger logger,
        string message, params object[] args)

    public static void LogWarning(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogWarning(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogWarning(this ILogger logger,
        Exception exception, string message, params object[] args)
    public static void LogWarning(this ILogger logger, string message,
        params object[] args)

    public static void LogError(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogError(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogError(this ILogger logger, Exception exception,
        string message, params object[] args)
    public static void LogError(this ILogger logger, string message,
        params object[] args)

    public static void LogCritical(this ILogger logger, EventId eventId,
        Exception exception, string message, params object[] args)
    public static void LogCritical(this ILogger logger, EventId eventId,
        string message, params object[] args)
    public static void LogCritical(this ILogger logger, Exception exception,
        string message, params object[] args)
    public static void LogCritical(this ILogger logger, string message,
        params object[] args)
```

```

public static void Log(this ILogger logger, LogLevel logLevel,
    string message, params object[] args)
public static void Log(this ILogger logger, LogLevel logLevel,
    EventId eventId, string message, params object[] args)
public static void Log(this ILogger logger, LogLevel logLevel,
    Exception exception, string message, params object[] args)
public static void Log(this ILogger logger, LogLevel logLevel,
    EventId eventId, Exception exception, string message,
    params object[] args)
}

```

Яркая характеристика ASP.NET Core связана с расширяемостью конвейера в целом и с ведением журнала в частности. Стандартное средство ведения журнала может быть заменено другой инфраструктурой ведения журнала при условии, что новая инфраструктура способна интегрироваться с установленным шаблоном ведения журнала. Serilog — одна из инфраструктур, которая хорошо интегрируется с ASP.NET Core. В последующих разделах демонстрируется создание инфраструктуры ведения журнала, основанной на Serilog, и конфигурирование приложений ASP.NET Core для использования нового кода регистрации в журнале.

Интерфейс IAppLogging

Начните с добавления в проект AutoLot.Services нового каталога по имени Logging. Добавьте в этот каталог новый файл под названием IAppLogging.cs для интерфейса IAppLogging<T>. Приведите содержимое файла IAppLogging.cs к следующему виду:

```

using System;
using System.Runtime.CompilerServices;
namespace AutoLot.Services.Logging
{
    public interface IAppLogging<T>
    {
        void LogAppError(Exception exception, string message,
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",
            [CallerLineNumber] int sourceLineNumber = 0);
        void LogAppError(string message,
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",
            [CallerLineNumber] int sourceLineNumber = 0);
        void LogAppCritical(Exception exception, string message,
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",
            [CallerLineNumber] int sourceLineNumber = 0);
        void LogAppCritical(string message,
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",
            [CallerLineNumber] int sourceLineNumber = 0);
        void LogAppDebug(string message,
            [CallerMemberName] string memberName = "",
            [CallerFilePath] string sourceFilePath = "",

```

```

[CallerLineNumber] int sourceLineNumber = 0);

void LogAppTrace(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0);

void LogAppInformation(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0);

void LogAppWarning(string message,
  [CallerMemberName] string memberName = "",
  [CallerFilePath] string sourceFilePath = "",
  [CallerLineNumber] int sourceLineNumber = 0);
}
}

```

Атрибуты `CallerMemberName`, `CallerFilePath` и `CallerLineNumber` инспектируют стек вызовов для получения имени члена, пути к файлу и номера строки в вызывающем коде. Например, если строка, в которой вызывается `LogAppWarning()`, находится в функции `DoWork()` внутри файла по имени `MyClassFile.cs`, а номер этой строки 36, тогда вызов:

```
_appLogger.LogAppWarning("A warning");
```

преобразуется в следующий эквивалент:

```
_appLogger.LogAppWarning("A warning",
  "DoWork", "c:/myfilepath/MyClassFile.cs", 36);
```

Если методу при вызове передаются значения, тогда переданные значения используются вместо значений из атрибутов.

Класс AppLogging

Класс `AppLogging` реализует интерфейс `IAppLogging`. Добавьте новый класс по имени `AppLogging` и модифицируйте операторы `using`, как показано ниже:

```

using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Serilog.Context;
```

Сделайте класс открытым и реализующим интерфейс `IAppLogging<T>`. Добавьте конструктор, который принимает экземпляр реализации `ILogger<T>` (интерфейс, поддерживаемый напрямую ASP.NET Core) и экземпляр реализации `IConfiguration`. Внутри конструктора получите доступ к конфигурации, чтобы извлечь имя приложения из файла настроек. Все три элемента (`ILogger<T>`, `IConfiguration` и имя приложения) необходимо сохранить в переменных уровня класса.

```

namespace AutoLot.Services.Logging
{
  public class AppLogging<T> : IAppLogging<T>
  {
```

```

private readonly ILogger<T> _logger;
private readonly IConfiguration _config;
private readonly string _applicationName;
public AppLogging(ILogger<T> logger, IConfiguration config)
{
    _logger = logger;
    _config = config;
    _applicationName = config.GetValue<string>("ApplicationName");
}
}
}
}

```

Инфраструктура Serilog позволяет добавлять свойства в стандартный процесс ведения журнала, заталкивая их внутрь LogContext. Добавьте внутренний метод для заталкивания свойств MemberName, FilePath, LineNumber и ApplicationName:

```

internal List<IDisposable> PushProperties(
    string memberName,
    string sourceFilePath,
    int sourceLineNumber)
{
    List<IDisposable> list = new List<IDisposable>
    {
        LogContext.PushProperty("MemberName", memberName),
        LogContext.PushProperty("FilePath", sourceFilePath),
        LogContext.PushProperty("LineNumber", sourceLineNumber),
        LogContext.PushProperty("ApplicationName", _applicationName)
    };
    return list;
}

```

Каждая реализация метода следует одному и тому же процессу. На первом шаге вызывается метод PushProperties () для добавления дополнительных свойств и затем соответствующий метод регистрации в журнале, предоставляемый LoggerExtensions в ILogger<T>. Ниже приведены все реализованные методы интерфейса:

```

public void LogAppError(Exception exception, string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogError(exception, message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}

public void LogAppError(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
}

```

```
var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
_logger.LogError(message);
foreach (var item in list)
{
    item.Dispose();
}
}

public void LogAppCritical(Exception exception, string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogCritical(exception, message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}

public void LogAppCritical(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogCritical(message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}

public void LogAppDebug(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogDebug(message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}

public void LogAppTrace(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogTrace(message);
    foreach (var item in list)
    {
```

```

        item.Dispose();
    }
}
public void LogAppInformation(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogInformation(message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}
public void LogAppWarning(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    var list = PushProperties(memberName, sourceFilePath, sourceLineNumber);
    _logger.LogWarning(message);
    foreach (var item in list)
    {
        item.Dispose();
    }
}

```

Конфигурация ведения журнала

Начните с замены стандартного средства ведения журнала инфраструктурой Serilog, добавив новый класс по имени `LoggingConfiguration` в каталог `Logging` проекта `AutoLot.Services`. Модифицируйте операторы `using` и сделайте класс открытым и статическим:

```

using System;
using System.Collections.Generic;
using System.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Serilog;
using Serilog.Events;
using Serilog.Sinks.MSSqlServer;
namespace AutoLot.Services.Logging
{
    public static class LoggingConfiguration
    {
    }
}

```

Для записи в различные целевые объекты для ведения журналов инфраструктура Serilog использует приемники (`sink`). Целевыми объектами, которые будут применяться для ведения журнала в приложениях ASP.NET Core, являются текстовый файл, база

данных и консоль. Приемники типа текстового файла и базы данных требуют конфигурации — выходного шаблона для текстового файла и списка полей для базы данных.

Чтобы настроить выходной шаблон, создайте следующее статическое строковое поле, допускающее только чтение:

```
private static readonly string OutputTemplate =
    @"[{Timestamp:yy-MM-dd HH:mm:ss} {Level}]{ApplicationName}:
{SourceContext}{NewLine} Message:{Message}{NewLine}in method
{MemberName} at {FilePath}:{LineNumber}{NewLine}{Exception}{NewLine}";
```

Приемник SQL Server нуждается в списке столбцов, идентифицированных с использованием типа `SqlColumn`. Добавьте показанный далее код для конфигурирования столбцов базы данных:

```
private static readonly ColumnOptions ColumnOptions = new ColumnOptions
{
    AdditionalColumns = new List<SqlColumn>
    {
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "ApplicationName"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "MachineName"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "MemberName"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "FilePath"},
        new SqlColumn {DataType = SqlDbType.Int, ColumnName = "LineNumber"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "SourceContext"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "RequestPath"},
        new SqlColumn {DataType = SqlDbType.VarChar,
                      ColumnName = "ActionName"},
    }
};
```

Замена стандартного средства ведения журнала вариантом `Serilog` представляет собой процесс из трех шагов. Первый шаг — очистка существующего поставщика, второй — добавление `Serilog` в `HostBuilder` и третий — завершение конфигурирования `Serilog`. Добавьте новый метод по имени `ConfigureSerilog()`, который является расширяющим методом для `IHostBuilder`:

```
public static IHostBuilder ConfigureSerilog(this IHostBuilder builder)
{
    builder
        .ConfigureLogging((context, logging) => { logging.ClearProviders(); })
        .UseSerilog((hostingContext, loggerConfiguration) =>
    {
        var config = hostingContext.Configuration;
        var connectionString =
            config.GetConnectionString("AutoLot").ToString();
        var tableName = config["Logging:MSSqlServer:tableName"].ToString();
        var schema = config["Logging:MSSqlServer:schema"].ToString();
        string restrictedToMinimumLevel =
            config["Logging:MSSqlServer:restrictedToMinimumLevel"].ToString();
```

```

if (!Enum.TryParse<LogEventLevel>(restrictedToMinimumLevel,
    out var logLevel))
{
    logLevel = LogEventLevel.Debug;
}
LogEventLevel level = (LogEventLevel)Enum.Parse(typeof(LogEventLevel),
restrictedToMinimumLevel);
var sqlOptions = new MSSqlServerSinkOptions
{
    AutoCreateSqlTable = false,
    SchemaName = schema,
    TableName = tableName,
};
if (hostingContext.HostingEnvironment.IsDevelopment())
{
    sqlOptions.BatchPeriod = new TimeSpan(0, 0, 0, 1);
    sqlOptions.BatchPostingLimit = 1;
}
loggerConfiguration
    .Enrich.FromLogContext()
    .Enrich.WithMachineName()
    .WriteTo.File(
        path: "ErrorLog.txt",
        rollingInterval: RollingInterval.Day,
        restrictedToMinimumLevel: logLevel,
        outputTemplate: OutputTemplate)
    .WriteTo.Console(restrictedToMinimumLevel: logLevel)
    .WriteTo.MSSqlServer(
        connectionString: connectionString,
        sqlOptions,
        restrictedToMinimumLevel: level,
        columnOptions: ColumnOptions);
});
return builder;
}

```

Теперь, когда все готово, пора заменить стандартное средство ведения журнала на Serilog.

Обновление настроек приложения

Раздел Logging во всех файлах настроек приложения (appsettings.json, appsettings.development.json и appsettings.production) для проектов AutoLot.Api и AutoLot.Dal потребуется модифицировать с учетом новой информации о ведении журнала и добавить имя приложения.

Откройте файлы appsettings.json и обновите разметку JSON, как показано ниже; удостоверьтесь в том, что применяете корректное имя проекта в узле ApplicationName и указываете строку подключения, соответствующую вашей системе:

```

// appsettings.json
{
    "Logging": {
        "MSSqlServer": {

```

```

    "schema": "Logging",
    "tableName": "SeriLogs",
    "restrictedToMinimumLevel": "Warning"
  }
},
"ApplicationName": "AutoLot.Api",
"AllowedHosts": "*"
}
// appsettings.development.json
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": false,
  "ApplicationName": "AutoLot.Api - Dev",
  "ConnectionStrings": {
    "AutoLot": {
      "Server": ".\,5433;Database=AutoLot;User ID=sa;Password=P@ssw0rd;"
    }
  }
}
// appsettings.production.json
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  },
  "RebuildDataBase": false,
  "ApplicationName": "AutoLot.Api - Prod",
  "ConnectionStrings": {
    "AutoLot": "It's a secret"
  }
}
}

```

Обновление Program.cs

Добавьте в файлы Program.cs в проектах AutoLot.Api и AutoLot.Mvc следующий оператор using:

```
using AutoLot.Services.Logging;
```

Модифицируйте метод CreateHostBuilder() в обоих проектах, как показано ниже:

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
  {
    webBuilder.UseStartup<Startup>();
  }).ConfigureSerilog();

```

Обновление Startup.cs

Добавьте в файлы Startup.cs в проектах AutoLot.Api и AutoLot.Mvc следующий оператор using:

```
using AutoLot.Services.Logging;
```

Затем необходимо поместить новые интерфейсы ведения журнала в контейнер DI. Добавьте в метод ConfigureServices() в обоих проектах такой код:

```
services.AddScoped(typeof(IAppLogging<>), typeof(AppLogging<>));
```

Обновление контроллера

Следующее обновление связано с заменой ссылок на ILogger ссылками на IAppLogging. Начните с класса WeatherForecastController в проекте AutoLot.Api. Добавьте в класс следующий оператор using:

```
using AutoLot.Services.Logging;
```

Далее измените ILogger<T> на IAppLogging<T>:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    ...
    private readonly IAppLogging<WeatherForecastController> _logger;
    public WeatherForecastController(IAppLogging<WeatherForecastController>
        logger)
    {
        _logger = logger;
    }
    ...
}
```

Теперь модифицируйте HomeController в проекте AutoLot.Mvc. Добавьте в класс следующий оператор using:

```
using AutoLot.Services.Logging;
```

Измените ILogger<T> на IAppLogging<T>:

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
    private readonly IAppLogging<HomeController> _logger;
    public HomeController(IAppLogging<HomeController> logger)
    {
        _logger = logger;
    }
    ...
}
```

После этого регистрация в журнале выполняется в каждом контроллере простым обращением к средству ведения журнала, например:

```
// WeatherForecastController.cs (AutoLot.Api)
[HttpGet]
```

```

public IEnumerable<WeatherForecast> Get()
{
    _logger.LogWarning("This is a test");
    ...
}

// HomeController.cs (AutoLot.Mvc)
[Route("/")]
[Route("/{controller}")]
[Route("/{controller}/{action}")]
[HttpGet]
public IActionResult Index()
{
    _logger.LogWarning("This is a test");
    return View();
}

```

Испытание инфраструктуры ведения журнала

Имея установленную инфраструктуру Serilog, самое время протестировать ведение журналов для приложений. Если вы используете Visual Studio, тогда укажите AutoLot.Mvc в качестве стартового проекта (щелкните правой кнопкой мыши на имени проекта в окне Solution Explorer, выберите в контекстном меню пункт Set as StartUp Project (Установить как стартовый проект) и щелкните на кнопке запуска с зеленой стрелкой или нажмите <F5>). В случае работы в VS Code откройте окно терминала (<Ctrl+>), перейдите в каталог AutoLot.Mvc и введите команду dotnet run.

В Visual Studio автоматически запустится браузер с представлением Home/Index. Если вы применяете VS Code, то вам понадобится открыть браузер и перейти по ссылке <https://localhost:5001>. После загрузки вы можете закрыть браузер, поскольку обращение к средству ведения журнала произошло при загрузке домашней страницы. Закрытие браузера в случае использования Visual Studio останавливает отладку. Чтобы остановить отладку в VS Code, нажмите <Ctrl+C> в окне терминала.

В каталоге проекта вы увидите файл по имени ErrorLogГГГГММДД.txt, в котором обнаружите запись, похожую на показанную ниже:

```

[ГГ-ММ-ДД чч:мм:сс Warning]AutoLot.Mvc -
  Dev:AutoLot.Mvc.Controllers.HomeController
Message:This is a test
in method Index at
D:\Projects\Books\csharp9-wf\Code\New\Chapter_29\AutoLot.Mvc\Controllers\
HomeController.cs:30

```

Для тестирования кода регистрации в журнале в проекте AutoLot.Api установите этот проект в качестве стартового (Visual Studio) или перейдите в каталог AutoLot.Api в окне терминала (VS Code). Нажмите <F5> или введите dotnet run и перейдите по ссылке <https://localhost:44375/swagger/index.html>. В итоге загрузится страница Swagger для приложения API (рис. 29.6).

Щелкните на кнопке GET для записи WeatherForecast. В результате откроется экран с деталями для этого метода действия, включая возможность Try it out (Опробовать), как видно на рис. 29.7.

После щелчка на кнопке Try it out щелкните на кнопке Execute (Выполнить), которая обеспечивает обращение к конечной точке (рис. 29.8).



Рис. 29.6. Начальная страница Swagger для AutoLot.Api

Code	Description	Links
200	Success	No links

Media type
text/plain

Example Value | Schema

```
[ { "Date": "2021-01-10T20:29:20.862Z", "TemperatureC": 0, "TemperatureF": 0, "Summary": "string" } ]
```

Рис. 29.7. Детальная информация для метода GET контроллера WeatherForecast

В каталоге проекта AutoLot.Api вы снова увидите файл по имени ErrorLog ГГГГММДД.txt и найдете в нем примерно такую запись:

```
[ГГ-ММ-ДД чч:мм:сс Warning]AutoLot.Api - Dev:AutoLot.Api.Controllers.
WeatherForecastController
Message:This is a test
in method Get at
D:\Projects\Books\csharp9-wf\Code\New\Chapter_29\AutoLot.Api\Controllers\
WeatherForecastController.cs:30
```

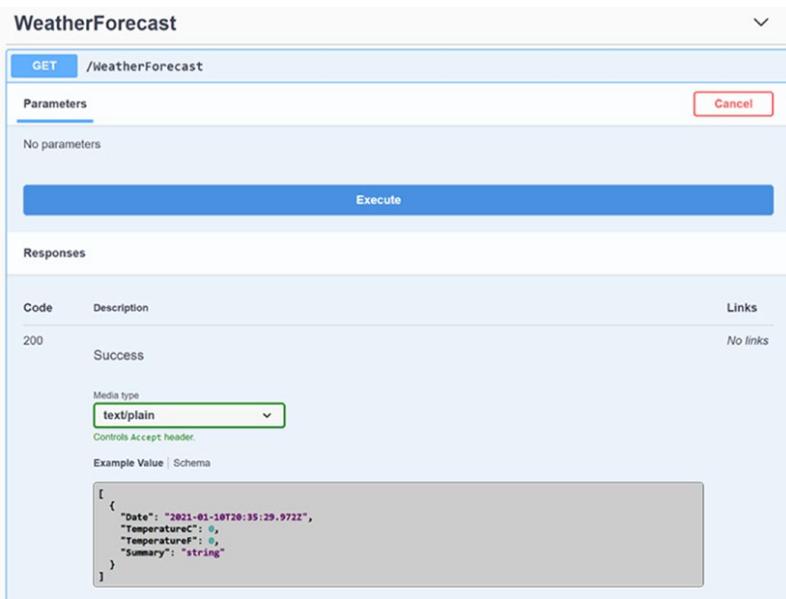


Рис. 29.8. Выполнение метода GET контроллера WeatherForecast

На заметку! Нововведением в версии ASP.NET Core 5 является то, что Swagger по умолчанию включается в шаблон API. Инструменты Swagger будут подробно исследованы в следующей главе.

Резюме

В главе была представлена инфраструктура ASP.NET Core. Глава начиналась с краткого обзора истории появления ASP.NET, после чего были рассмотрены функциональные средства из классических инфраструктур ASP.NET MVC и ASP.NET Web API, которые присутствуют в ASP.NET Core.

Далее вы узнали о новых средствах ASP.NET Core и о том, как они работают. После изучения различных способов запуска и отладки приложений ASP.NET Core вы создали решение с двумя проектами ASP.NET Core — для общей библиотеки прикладных служб и для уровня доступа к данным AutoLot (из главы 23). Наконец, вы заменили в обоих проектах стандартное средство ведения журнала ASP.NET Core инфраструктурой Serilog.

В следующей главе приложение AutoLot.Api будет завершено.

ГЛАВА 30

Создание служб REST с помощью ASP.NET Core

В предыдущей главе была представлена инфраструктура ASP.NET Core, обсуждались ее новые возможности, были созданы проекты, а также обновлен код в AutoLot.Mvc и AutoLot.Api для включения AutoLot.Dal и ведения журнала Serilog. Внимание в текущей главе будет сосредоточено на завершении работы над REST-службой AutoLot.Api.

На заметку! Исходный код, рассматриваемый в этой главе, находится в папке Chapter_30 внутри хранилища GitHub для настоящей книги. Вы также можете продолжить работу с решением, начатым в главе 29.

Введение в REST-службы ASP.NET Core

Инфраструктура ASP.NET MVC начала набирать обороты почти сразу после своего выхода, а в составе версий ASP.NET MVC 4 и Visual Studio 2012 компания Microsoft выпустила ASP.NET Web API. Версия ASP.NET Web API 2 вышла вместе с Visual Studio 2013 и затем с выходом Visual Studio 2013 Update 1 была модернизирована до версии 2.2.

Продукт ASP.NET Web API с самого начала разрабатывался как основанная на службах инфраструктура для построения служб REST (REpresentational State Transfer — передача состояния представления), которая базируется на инфраструктуре MVC минус “V” (представление) с рядом оптимизаций, направленных на создание автономных служб. Такие службы могут вызываться с применением любой технологии, а не только тех, которые производит Microsoft. Обращения к службе Web API основаны на базовых HTTP-методах (GET, PUT, POST, DELETE) и осуществляются через универсальный идентификатор ресурса (uniform resource identifier — URI), например:

```
http://www.skimedic.com:5001/api/cars
```

Он похож на унифицированный указатель ресурса (uniform resource locator — URL), поскольку таковым и является! Указатель URL — это просто идентификатор URI, который указывает на физический ресурс в сети.

При вызове служб Web API используется схема HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста) на конкретном хосте (в приведенном выше примере www.skimedic.com) и специфическом порте (5001), за которыми указывается путь (api/cars), а также необязательные запрос и фрагмент (в примере отсутствуют). Обращение к службе Web API может также содержать текст в теле сообщения, как вы увидите далее в этой главе. Из предыдущей главы вы узнали, что ASP.NET Core объединяет Web API и MVC в одну инфраструктуру.

Создание действий контроллера с использованием служб REST

Вспомните, что действия возвращают тип `IActionResult` (или `Task<IActionResult>` для асинхронных операций). Кроме вспомогательных методов в `ControllerBase`, возвращающих специфические коды состояния HTTP методы действий способны возвращать содержимое как ответы в формате JSON (JavaScript Object Notation — запись объектов JavaScript).

На заметку! Стого говоря, методы действий могут возвращать широкий диапазон форматов. Формат JSON рассматривается в книге из-за своей популярности.

Результаты ответов в формате JSON

Большинство служб REST получают и отправляют данные клиентам с применением формата JSON. Ниже приведен простой пример данных JSON, состоящих из двух значений:

```
[  
    "value1",  
    "value2"  
]
```

На заметку! Сериализация JSON с использованием `System.Text.Json` подробно обсуждалась в главе 20.

Службы API также применяют коды состояния HTTP для сообщения об успехе или неудаче. Некоторые вспомогательные методы для возвращения кодов состояния HTTP, доступные в классе `ControllerBase`, были перечислены в табл. 29.3. Успешные запросы возвращают коды состояния в диапазоне до 200, причем 200 (OK) является самым распространенным кодом успеха. В действительности он настолько распространен, что вам не придется возвращать его явно. Если никаких исключений не возникало, а код состояния не был указан, тогда клиенту будет возвращен код 200 вместе с любыми данными.

Чтобы подготовиться к последующим примерам, создайте в проекте `AutoLot.Api` новый контроллер, добавив в каталог `Controllers` новый файл по имени `ValuesController.cs` с показанным ниже кодом:

```
using System.Collections.Generic;  
using Microsoft.AspNetCore.Mvc;  
  
[Route("api/[controller]")]  
[ApiController]  
public class ValuesController : ControllerBase  
{  
}
```

На заметку! В среде Visual Studio для контроллеров предусмотрены шаблоны. Чтобы получить к ним доступ, щелкните правой кнопкой мыши на имени каталога `Controllers` в проекте `AutoLot.Api`, выберите в контекстном меню пункт `Add⇒Controller` (`Добавить⇒Контроллер`) и укажите шаблон `MVC Controller — Empty` (`Контроллер MVC — Пустой`).

В коде устанавливается маршрут для контроллера с использованием значения `(api)` и маркера `[[controller]]`. Такой шаблон маршрута будет соответствовать URL наподобие `www.skimedc.com/api/values`. Атрибут `ApiController` выбирает несколько специфичных для API средств (раскрываются в следующем разделе). Наконец, класс контроллера наследуется от `ControllerBase`. Как обсуждалось в главе 29, в инфраструктуре ASP.NET Core все типы контроллеров, доступные в классической версии ASP.NET, были объединены в один класс по имени `Controller` с базовым классом `ControllerBase`. Класс `Controller` обеспечивает функциональность, специфичную для представлений ("V" в MVC), тогда как `ControllerBase` предлагает оставшуюся базовую функциональность для приложений в стиле MVC.

Существует несколько способов возвращения содержимого в формате JSON из метода действия. Все приведенные далее примеры возвращают те же самые данные JSON с кодом состояния 200. Различия практически полностью стилистические. Добавьте в свой класс `ValuesController` следующий код:

```
[HttpGet]
public IActionResult Get()
{
    return Ok(new string[] { "value1", "value2" });
}

[HttpGet("one")]
public IEnumerable<string> Get1()
{
    return new string[] { "value1", "value2" };
}

[HttpGet("two")]
public ActionResult<IEnumerable<string>> Get2()
{
    return new string[] { "value1", "value2" };
}

[HttpGet("three")]
public string[] Get3()
{
    return new string[] { "value1", "value2" };
}

[HttpGet("four")]
public IActionResult Get4()
{
    return new JsonResult(new string[] { "value1", "value2" });
}
```

Чтобы протестировать код, запустите приложение `AutoLot.Api`; вы увидите список всех методов из `ValuesController` в пользовательском интерфейсе (рис. 30.1). Вспомните, что при определении маршрутов суффикс `Controller` отбрасывается из имен маршрутов, поэтому конечные точки в `ValuesController` сопоставляются с `Values`, а не с `ValuesController`.

Для выполнения одного из методов щелкните на кнопке `GET`, на кнопке `Try it out` (Опробовать) и на кнопке `Execute` (Выполнить). После выполнения метода пользовательский интерфейс обновится, чтобы отобразить результаты; наиболее важная часть пользовательского интерфейса `Swagger` показана на рис. 30.2.

Вы увидите, что выполнение каждого метода приводит к получению тех же самых результатов JSON.

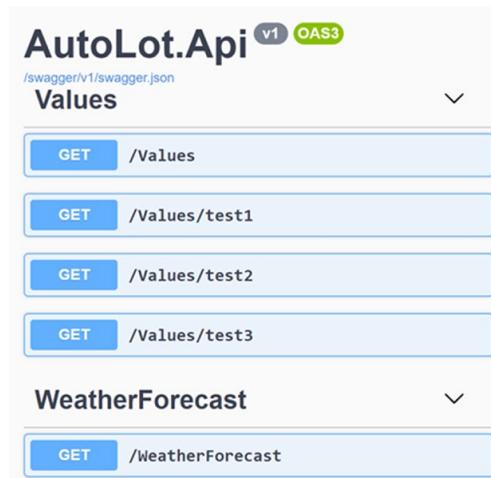


Рис. 30.1. Документирующая страница Swagger

Code	Details
200	<p>Response body</p> <pre>["value1", "value2"]</pre> <p>Download</p> <p>Response headers</p> <pre>content-type: application/json; charset=utf-8 date: Sun, 10 Jan 2021 21:28:42 GMT server: Microsoft-IIS/10.0 x-powered-by: ASP.NET</pre>

Рис. 30.2. Информация ответа сервера в Swagger

Атрибут ApiController

Атрибут ApiController, появившийся в версии ASP.NET Core 2.1, в сочетании с классом ControllerBase обеспечивает правила, соглашения и линии поведения, специфичные для REST. Соглашения и линии поведения рассматриваются в последующих разделах.

Обязательность маршрутизации с помощью атрибутов

При наличии атрибута ApiController контроллер обязан использовать маршрутизацию с помощью атрибутов. Это просто принудительное применение того, что многие расценивают как установленную практику.

Автоматические ответы с кодом состояния 400

Если есть проблема с привязкой модели, то действие будет автоматически возвращать код состояния HTTP 400 (Bad Request), что заменяет следующий код:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

Для выполнения показанной выше проверки инфраструктура ASP.NET Core использует фильтр действий ModelStateInvalidFilter. При наличии ошибок привязки или проверки достоверности ответ HTTP 400 в своем теле содержит детальные сведения об ошибках. Вот пример:

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "|7fb5e16a-4c8f23bbfc974667.",
    "errors": {
        "": [
            "A non-empty request body is required."
        ]
    }
}
```

Такое поведение можно отключить через конфигурацию в методе ConfigureServices() класса Startup:

```
services.AddControllers().ConfigureApiBehaviorOptions(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});
```

Выведение источников для привязки параметров

Механизм привязки моделей будет логически выводить источники извлечения значений на основе соглашений, описанных в табл. 30.1.

Таблица 30.1. Соглашения о выводении источников для привязки

Источник	Привязываемые параметры
FromBody	Выводится для параметров сложных типов кроме встроенных типов со специальным смыслом, таких как IFormCollection или CancellationToken. Может существовать только один параметр FromBody, иначе сгенерируется исключение. Если привязка требуется для параметра простого типа (скажем, string либо int), то атрибут FromBody по-прежнему обязательен
FromForm	Выводится для параметров действий типов IFormFile и IFormFileCollection. Когда параметр помечен с помощью FromForm, выводится тип содержимого multipart/form-data
FromRoute	Выводится для любого параметра, имя которого совпадает с именем маркера маршрута
FromQuery	Выводится для любых других параметров действий

Такое поведение можно отключить через конфигурацию в методе `ConfigureServices()` класса `Startup`:

```
services.AddControllers().ConfigureApiBehaviorOptions(options =>
{
    // Подавить все выведение источников для привязки.
    options.SuppressInferBindingSourcesForParameters = true;

    // Подавить выведение типа содержимого multipart/form-data.
    options.SuppressConsumesConstraintForFormFileParameters = true;
});
```

Детальные сведения о проблемах для кодов состояния ошибок

ASP.NET Core трансформирует результат ошибки (состояние 400 или выше) в результат с помощью типа `ProblemDetails`, который показан ниже:

```
public class ProblemDetails
{
    public string Type { get; set; }
    public string Title { get; set; }
    public int? Status { get; set; }
    public string Detail { get; set; }
    public string Instance { get; set; }
    public IDictionary<string, object> Extensions { get; }
        = new Dictionary<string, object>(StringComparer.OrdinalIgnoreCase);
}
```

Чтобы протестировать это поведение, добавьте в `ValuesController` еще один метод:

```
[HttpGet("error")]
public IActionResult Error()
{
    return NotFound();
}
```

Запустите приложение и посредством пользовательского интерфейса Swagger выполните новую конечную точку `error`. Результатом по-прежнему будет код состояния 404 (Not Found), но в теле ответа возвратится дополнительная информация. Ниже приведен пример ответа (ваше значение `traceId` будет другим):

```
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
    "title": "Not Found",
    "status": 404,
    "traceId": "00-9a609e7e05f46d4d82d5f897b90da624-a6484fb34a7d3a44-00"
}
```

Такое поведение можно отключить через конфигурацию в методе `ConfigureServices()` класса `Startup`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
{
    options.SuppressMapClientErrors = true;
});
```

Когда поведение отключено, вызов конечной точки `error` возвращает код состояния 404 без какой-либо дополнительной информации.

Обновление настроек Swagger/OpenAPI

Продукт Swagger (также известный как OpenAPI) является стандартом с открытым кодом для документирования служб REST, основанных на API. Два главных варианта для добавления Swagger к API-интерфейсам ASP.NET Core — Swashbuckle и NSwag. Версия ASP.NET Core 5 теперь включает Swashbuckle в виде части шаблона нового проекта. Документация `swagger.json`, сгенерированная для `AutoLot.Api`, содержит информацию по сайту, по каждой конечной точке и по любым объектам, задействованным в конечных точках.

Пользовательский интерфейс Swagger базируется на веб-интерфейсе и позволяет интерактивным образом исследовать конечные точки приложения, а также тестиировать их (как делалось ранее в этой главе). Его можно расширить, добавляя документацию в сгенерированный файл `swagger.json`.

Обновление обращений к Swagger в классе Startup

Стандартный шаблон проекта API добавляет код для генерирования файла `swagger.json` в метод `ConfigureService()` класса `Startup`:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "AutoLot.Api",
                                            Version = "v1" });
});
```

Первое изменение стандартного кода предусматривает добавление метаданных к `OpenApiInfo`. Модифицируйте вызов `AddSwaggerGen()` следующим образом, чтобы обновить заголовок и добавить описание и сведения о лицензии:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new OpenApiInfo
        {
            Title = "AutoLot Service",
            Version = "v1",
            Description = "Service to support the AutoLot dealer site",
            License = new OpenApiLicense
            {
                Name = "Skimedic Inc",
                Url = new Uri("http://www.skimedic.com")
            }
        });
});
```

Следующий шаг связан с переносом вызовов `UseSwagger()` и `UseSwaggerUI()` из блока, предназначенного только для среды разработки, в главный путь выполнения внутри метода `Configure()`. Кроме того, поменяйте заголовок "AutoLot.Api v1" на "AutoLot Service v1".

```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    ApplicationDbContext context)
{
    if (env.IsDevelopment())
    {
        // Если среда разработки, тогда отображать отладочную информацию.
        app.UseDeveloperExceptionPage();
        // Первоначальный код:
        // app.UseSwagger();
        // app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
        //                                             "AutoLot.Api v1"));
        // Инициализировать базу данных.
        if (Configuration.GetValue<bool>("RebuildDataBase"))
        {
            SampleDataInitializer.ClearAndReseedDatabase(context);
        }
    }

    // Включить промежуточное ПО для обслуживания сгенерированного
    // файла Swagger как конечной точки JSON.
    app.UseSwagger();

    // Включить промежуточное ПО для обслуживания пользовательского
    // интерфейса Swagger (HTML, JS, CSS и т.д.), указывая конечную
    // точку JSON для Swagger.
    app.UseSwaggerUI(c => { c.SwaggerEndpoint("/swagger/v1/swagger.json",
                                                "AutoLot Service v1"); });

    ...
}

```

В предыдущем коде используется `Swagger (app.UseSwagger())` и пользовательский интерфейс `Swagger (app.UseSwaggerUI())`. В нем также конфигурируется конечная точка для файла `swagger.json`.

Добавление файла XML-документации

Инфраструктура .NET Core способна генерировать файл XML-документации для вашего проекта, исследуя методы на предмет наличия комментариев с тремя символами прямой косой черты (///). Чтобы включить такую функциональность в Visual Studio, щелкните правой кнопкой мыши на имени проекта `AutoLot.Api` и в контекстном меню выберите пункт `Properties` (Свойства). В открывшемся диалоговом окне `Properties` (Свойства) перейдите на вкладку `Build` (Сборка), отметьте флажок `XML documentation file` (Файл XML-документации) и укажите в качестве имени файла `AutoLot.Api.xml`. Кроме того, введите 1591 в текстовом поле `Suppress warnings` (Подавлять предупреждения), как показано на рис. 30.3.

Те же настройки можно вводить прямо в файле проекта. Ниже показан раздел `PropertyGroup`, который понадобится добавить:

```

<PropertyGroup Condition=" '$(Configuration)|$(Platform)'=='Debug|AnyCPU' >
    <DocumentationFile>AutoLot.Api.xml</DocumentationFile>
    <NoWarn>1701;1702;1591;</NoWarn>
</PropertyGroup>

```

Настройка `NoWarn` с указанием 1591 отключает выдачу предупреждений компилятором для методов, которые не имеют XML-комментариев.

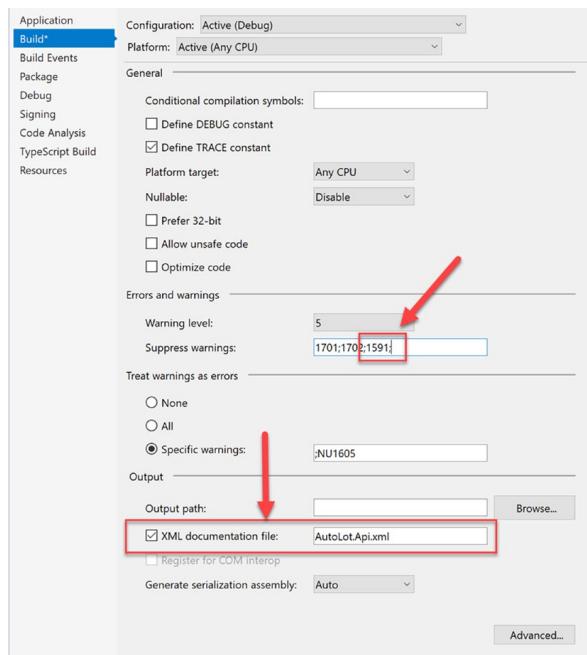


Рис. 30.3. Добавление файла XML-документации и подавление предупреждения 1591

На заметку! Предупреждения 1701 и 1702 являются пережитками ранних дней классической платформы .NET, которые обнаруживают компиляторы .NET Core.

Чтобы взглянуть на процесс в действии, модифицируйте метод `Get()` класса `ValuesController` следующим образом:

```
/// <summary>
/// This is an example Get method returning JSON
/// </summary>
/// <remarks>This is one of several examples for returning JSON:
/// <pre>
/// [
///   "value1",
///   "value2"
/// ]
/// </pre>
/// </remarks>
/// <returns>List of strings</returns>
[HttpGet]
public IActionResult Get()
{
    return Ok(new string[] { "value1", "value2" });
}
```

Когда вы скомпилируете проект, в корневом каталоге проекта появится новый файл по имени AutoLot.Api.xml. Открыв его, вы увидите только что добавленные комментарии:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>AutoLot.Api</name>
  </assembly>
  <members>
    <member name="M:AutoLot.Api.Controllers.ValuesController.Get">
      <summary>
        This is an example Get method returning JSON
      </summary>
      <remarks>This is one of several examples for returning JSON:
      <pre>
      [
        "value1",
        "value2"
      ]
      </pre>
    </remarks>
    <returns>List of strings</returns> </member>
  </members>
</doc>
```

На заметку! Если вы вводите три символа прямой косой черты перед определением класса или метода в Visual Studio, то среда создает начальную заглушки для XML-комментариев.

Далее необходимо объединить XML-комментарии со сгенерированным файлом swagger.json.

Добавление XML-комментариев в процесс генерации Swagger

Сгенерированные XML-комментарии должны быть добавлены в процесс генерации swagger.json. Начните с добавления следующих операторов using в файл класса Startup.cs:

```
using System.IO;
using System.Reflection;
```

Файл XML-документации добавляется в Swagger вызовом метода `IncludeXmlComments()` внутри метода `AddSwaggerGen()`.

Перейдите к методу `ConfigureServices()` класса `Startup` и модифицируйте метод `AddSwaggerGen()`, добавив файл XML-документации:

```
services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1",
    new OpenApiInfo
    {
      Title = "AutoLot Service",
      Version = "v1",
      Description = "Service to support the AutoLot dealer site",
      License = new OpenApiLicense
    }
```

```

        Name = "Skimedic Inc",
        Url = new Uri("http://www.skimedic.com")
    }
});
var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
var xmlPath = Path.Combine(ApplicationContext.BaseDirectory, xmlFile);
c.IncludeXmlComments(xmlPath);
});
}

```

Запустите приложение и загляните в пользовательский интерфейс Swagger. Обратите внимание на XML-комментарии, интегрированные в пользовательский интерфейс Swagger (рис. 30.4).

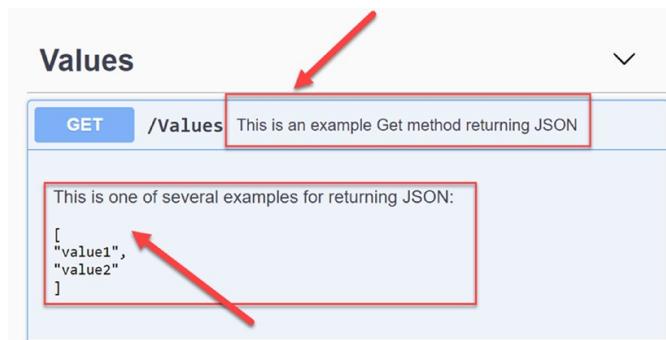


Рис. 30.4. XML-документация, интегрированная в пользовательский интерфейс Swagger

Помимо XML-документации документирование может быть улучшено дополнительной конфигурацией конечных точек приложения.

Дополнительные возможности документирования для конечных точек API

Существуют дополнительные атрибуты, которые дополняют документацию Swagger. Чтобы применить их, начните с добавления показанных далее операторов `using` в файл `ValuesController.cs`:

```
using Microsoft.AspNetCore.Http;
using Swashbuckle.AspNetCore.Annotations;
```

Атрибут `Produces` задает тип содержимого для конечной точки. Атрибут `ProducesResponseType` использует перечисление `StatusCodes` для указания возможного кода возврата для конечной точки. Модифицируйте метод `Get()` класса `ValuesController`, чтобы установить `application/json` в качестве возвращаемого типа и сообщить о том, что результатом действия будет либо 200 (OK), либо 400 (Bad Request):

```
[HttpGet]
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<IEnumerable<string>> Get()
{
    return new string[] {"value1", "value2"};
}
```

Хотя атрибут `ProducesResponseType` добавляет в документацию коды ответов, настроить эту информацию невозможно. К счастью, Swashbuckle добавляет атрибут `SwaggerResponse`, предназначенный как раз для такой цели. Приведите код метода `Get()` к следующему виду:

```
[HttpGet]
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(400, "The request was invalid")]
public ActionResult<IEnumerable<string>> Get()
{
    return new string[] {"value1", "value2"};
}
```

Прежде чем аннотации Swagger будут приняты и добавлены в генерированную документацию, их потребуется включить. Откройте файл `Startup.cs` и перейдите к методу `Configure()`. Обновите вызов `AddSwaggerGen()`, как показано ниже:

```
services.AddSwaggerGen(c =>
{
    c.EnableAnnotations();
    ...
});
```

Теперь, просматривая раздел ответов в пользовательском интерфейсе Swagger, вы будете видеть настроенный обмен сообщениями (рис. 30.5).

The screenshot shows the 'Responses' section of the Swagger UI. It lists two entries: '200' and '400'. The '200' entry is highlighted with a red box. It contains the description 'The execution was successful', a dropdown menu set to 'application/json' (also highlighted with a green box), and a code example box containing '["string"]'. The '400' entry is also highlighted with a red box. It contains the description 'The request was invalid', a dropdown menu set to 'application/json', and a code example box containing a JSON schema object with fields like 'type', 'title', 'status', 'detail', and 'instance'.

Responses		
Code	Description	Links
200	The execution was successful Media type application/json <small>Controls Accept header.</small> Example Value Schema <pre>["string"]</pre>	No links
400	The request was invalid Media type application/json <small>Controls Accept header.</small> Example Value Schema <pre>{ "type": "string", "title": "string", "status": 0, "detail": "string", "instance": "string" }</pre>	No links

Рис. 30.5. Обновленные ответы в пользовательском интерфейсе Swagger

На заметку! В Swashbuckle поддерживается большой объем дополнительной настройки, за сведениями о которой обращайтесь в документацию по ссылке <https://github.com/domaindrivendev/Swashbuckle.AspNetCore>.

Построение методов действий API

Большинство функциональных средств приложения AutoLot.Api можно отнести к одному из перечисленных далее методов:

- GetOne()
- GetAll()
- UpdateOne()
- AddOne()
- DeleteOne()

Основные методы API будут реализованы в обобщенном базовом контроллере API. Начните с создания нового каталога под названием `Base` в каталоге `Controllers` проекта `AutoLot.Api`. Добавьте в этот каталог новый файл класса по имени `BaseCrudController.cs`. Модифицируйте операторы `using` и определение класса, как демонстрируется ниже:

```
using System;
using System.Collections.Generic;
using AutoLot.Dal.Exceptions;
using AutoLot.Models.Entities.Base;
using AutoLot.Dal.Repos.Base;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Swashbuckle.AspNetCore.Annotations;

namespace AutoLot.Api.Controllers.Base
{
    [ApiController]
    public abstract class BaseCrudController<T, TController> : ControllerBase
        where T : BaseEntity, new()
        where TController : BaseCrudController<T, TController>
    {
    }
}
```

Класс является открытым и абстрактным, а также унаследованным от `ControllerBase`. Он принимает два обобщенных параметра типа. Первый тип ограничивается так, чтобы быть производным от `BaseEntity` и иметь стандартный конструктор, а второй — быть производным от `BaseCrudController` (для представления производных контроллеров). Когда к базовому классу добавляется атрибут `ApiController`, производные контроллеры получают функциональность, обеспечиваемую атрибутом.

На заметку! Для этого класса не определен маршрут. Он будет установлен с использованием производных классов.

Конструктор

На следующем шаге добавляются две защищенные переменные уровня класса: одна для хранения реализации интерфейса `IRepo<T>` и еще одна для хранения реализации интерфейса `IAppLogging<TController>`. Обе они должны устанавливаться с применением конструктора.

```
protected readonly IRepo<T> MainRepo;
protected readonly IAppLogging<TController> Logger;
protected BaseCrudController(IRepo<T> repo,
                            IAppLogging<TController> logger)
{
    MainRepo = repo;
    Logger = logger;
}
```

Методы `GetXXX()`

Есть два HTTP-метода GET, `GetOne()` и `GetAll()`. Оба они используют хранилище, переданное контроллеру. Первым делом добавьте метод `GetAll()`, который служит в качестве конечной точки для шаблона маршрута контроллера:

```
/// <summary>
/// Gets all records
/// </summary>
/// <returns>All records</returns>
/// <response code="200">Returns all items</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(400, "The request was invalid")]
[HttpGet]
public ActionResult<IEnumerable<T>> GetAll()
{
    return Ok(MainRepo.GetAllIgnoreQueryFilters());
}
```

Следующий метод получает одиночную запись на основе параметра `id`, который передается как обязательный параметр маршрута и добавляется к маршруту производного контроллера:

```
/// <summary>
/// Gets a single record
/// </summary>
/// <param name="id">Primary key of the record</param>
/// <returns>Single record</returns>
/// <response code="200">Found the record</response>
/// <response code="204">No content</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(204, "No content")]
[HttpGet("{id}")]
```

```
public ActionResult<T> GetOne(int id)
{
    var entity = MainRepo.Find(id);
    if (entity == null)
    {
        return NotFound();
    }
    return Ok(entity);
}
```

Значение маршрута автоматически присваивается параметру id (неявно из [FromRoute]).

Метод UpdateOne()

Обновление записи делается с применением HTTP-метода PUT. Ниже приведен код метода UpdateOne():

```
/// <summary>
/// Updates a single record
/// </summary>
/// <remarks>
/// Sample body:
/// <pre>
/// {
///     "Id": 1,
///     "TimeStamp": "AAAAAAAAB+E="
///     "MakeId": 1,
///     "Color": "Black",
///     "PetName": "Zippy",
///     "MakeColor": "VW (Black)",
/// }
/// </pre>
/// </remarks>
/// <param name="id">Primary key of the record to update</param>
/// <returns>Single record</returns>
/// <response code="200">Found and updated the record</response>
/// <response code="400">Bad request</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(400, "The request was invalid")]
[HttpPut("{id}")]
public IActionResult UpdateOne(int id, T entity)
{
    if (id != entity.Id)
    {
        return BadRequest();
    }
    try
    {
        MainRepo.Update(entity);
    }
```

```

    catch (CustomException ex)
    {
        // Пример специального исключения.
        // Должно обрабатываться более элегантно.
        return BadRequest(ex);
    }
    catch (Exception ex)
    {
        // Должно обрабатываться более элегантно.
        return BadRequest(ex);
    }
    return Ok(entity);
}

```

Метод начинается с установки маршрута как запроса `HttpPut` на основе маршрута производного контроллера с обязательным параметром маршрута `id`. Значение маршрута присваивается параметру `id` (неявно из `[FromRoute]`), а сущность (`entity`) извлекается из тела запроса (неявно из `[FromBody]`). Также обратите внимание, что проверка достоверности модели отсутствует, поскольку делается автоматически атрибутом `ApiController`. Если состояние модели укажет на наличие ошибок, тогда клиенту будет возвращен код 400 (Bad Request).

Метод проверяет, совпадает ли значение маршрута (`id`) со значением `id` в теле запроса. Если не совпадает, то возвращается код 400 (Bad Request). Если совпадает, тогда используется хранилище для обновления записи. Если обновление терпит неудачу с генерацией исключения, то клиенту возвращается код 400 (Bad Request). Если операция обновления проходит успешно, тогда клиенту возвращается код 200 (OK) и обновленная запись в качестве тела запроса.

На заметку! Обработка исключений в этом примере (а также в остальных примерах) абсолютно неадекватна. В производственных приложениях вы должны задействовать все знания, полученные к настоящему времени, чтобы элегантно обрабатывать возникающие проблемы в соответствии с имеющимися требованиями.

Метод `AddOne ()`

Вставка записи делается с применением HTTP-метода `POST`. Ниже приведен код метода `AddOne ()`:

```

/// <summary>
/// Adds a single record
/// </summary>
/// <remarks>
/// Sample body:
/// <pre>
/// {
///     "Id": 1,
///     "TimeStamp": "AAAAAAAAB+E=",
///     "MakeId": 1,
///     "Color": "Black",
///     "PetName": "Zippy",
///     "MakeColor": "VW (Black)",
/// }

```

```

/// </pre>
/// </remarks>
/// <returns>Added record</returns>
/// <response code="201">Found and updated the record</response>
/// <response code="400">Bad request</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[SwaggerResponse(201, "The execution was successful")]
[SwaggerResponse(400, "The request was invalid")]
[HttpPost]
public ActionResult<T> AddOne(T entity)
{
    try
    {
        MainRepo.Add(entity);
    }
    catch (Exception ex)
    {
        return BadRequest(ex);
    }
    return CreatedAtAction(nameof(GetOne), new { id = entity.Id}, entity);
}

```

Метод начинается с определения маршрута как запроса `HttpPost`. Параметр маршрута отсутствует, потому что создается новая запись. Если хранилище успешно добавит запись, то ответом будет результат вызова метода `CreatedAtAction()`, который возвращает клиенту код 201 вместе с URL для вновь созданной сущности в виде значения заголовка `Location`. Вновь созданная сущность в формате JSON помещается внутрь тела ответа.

Метод DeleteOne ()

Удаление записи делается с применением HTTP-метода `DELETE`. Ниже приведен код метода `DeleteOne()`:

```

/// <summary>
/// Deletes a single record
/// </summary>
/// <remarks>
/// Sample body:
/// <pre>
/// {
///     "Id": 1,
///     "TimeStamp": "AAAAAAAAB+E="
/// }
/// </pre>
/// </remarks>
/// <returns>Nothing</returns>
/// <response code="200">Found and deleted the record</response>
/// <response code="400">Bad request</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]

```

```
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(400, "The request was invalid")]
[HttpDelete("{id}")]
public ActionResult<T> DeleteOne(int id, T entity)
{
    if (id != entity.Id)
    {
        return BadRequest();
    }
    try
    {
        MainRepo.Delete(entity);
    }
    catch (Exception ex)
    {
        // Должно обрабатываться более элегантно.
        return new BadRequestObjectResult(ex.GetBaseException()?.Message);
    }
    return Ok();
}
```

Метод начинается с определения маршрута как запроса `HttpDelete` с обязательным параметром маршрута `id`. Значение `id` в маршруте сравнивается со значением `id`, отправленным с остальной частью сущности в теле запроса, и если они не совпадают, то возвращается код 400 (Bad Request). Если хранилище успешно удаляет запись, тогда клиенту возвращается код 200 (OK), а если возникла какая-то ошибка, то клиент получает код 400 (Bad Request).

На этом создание базового контроллера завершено.

Класс CarsController

Приложению `AutoLot.Api` необходим дополнительный метод `HttpGet` для получения записей `Car` на основе значения `Make`. Он будет создан в новом классе по имени `CarsController`. Создайте в каталоге `Controllers` новый пустой контроллер API под названием `CarsController`. Модифицируйте операторы `using` следующим образом:

```
using System.Collections.Generic;
using AutoLot.Api.Controllers.Base;
using Microsoft.AspNetCore.Mvc;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Http;
using Swashbuckle.AspNetCore.Annotations;
```

Класс `CarsController` является производным от класса `BaseCrudController` и определяет маршрут на уровне контроллера. Конструктор принимает специфичное для сущности хранилище и средство ведения журнала. Вот первоначальный код контроллера:

```
namespace AutoLot.Api.Controllers
{
    [Route("api/[controller]")]
}
```

```

public class CarsController : BaseCrudController<Car, CarsController>
{
    public CarsController(ICarRepo carRepo,
                          IAppLogging<CarsController> logger) :
        base(carRepo, logger)
    {
    }
}
}

```

Класс CarsController расширяет базовый класс еще одним методом действия, который получает все записи об автомобилях конкретного производителя. Добавьте показанный ниже код:

```

/// <summary>
/// Gets all cars by make
/// </summary>
/// <returns>All cars for a make</returns>
/// <param name="id">Primary key of the make</param>
/// <response code="200">Returns all cars by make</response>
[Produces("application/json")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[SwaggerResponse(200, "The execution was successful")]
[SwaggerResponse(204, "No content")]

[HttpGet("bymake/{id?}")]
public ActionResult<IEnumerable<Car>> GetCarsByMake(int? id)
{
    if (id.HasValue && id.Value > 0)
    {
        return Ok(((ICarRepo)MainRepo).GetAllBy(id.Value));
    }
    return Ok(MainRepo.GetAllIgnoreQueryFilters());
}

```

Атрибут `HttpGet` расширяет маршрут константой `bymake` и необязательным идентификатором производителя для фильтрации, например:

`https://localhost:5021/api/cars/bymake/5`

Сначала в методе проверяется, было ли передано значение для `id`. Если нет, то получаются все автомобили. Если значение было передано, тогда с использованием метода `GetAllBy()` класса `CarRepo` получаются автомобили по производителю. Поскольку защищенное свойство `MainRepo` базового класса определено с типом `IRepo<T>`, его потребуется привести к типу `ICarRepo`.

Оставшиеся контроллеры

Все оставшиеся контроллеры, специфичные для сущностей, будут производными от класса `BaseCrudController`, но без добавления дополнительной функциональности. Добавьте в каталог `Controllers` еще четыре пустых контроллера API с именами `CreditRisksController`, `CustomersController`, `MakesController` и `OrdersController`.

Вот код оставшихся контроллеров:

```
// CreditRisksController.cs
using AutoLot.Api.Controllers.Base;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Mvc;
namespace AutoLot.Api.Controllers
{
    [Route("api/[controller]")]
    public class CreditRisksController
        : BaseCrudController<CreditRisk, CreditRisksController>
    {
        public CreditRisksController(
            ICreditRiskRepo creditRiskRepo,
            IAppLogging<CreditRisksController> logger)
            : base(creditRiskRepo, logger)
        {
        }
    }
}

// CustomersController.cs
using AutoLot.Api.Controllers.Base;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Mvc;
namespace AutoLot.Api.Controllers
{
    [Route("api/[controller]")]
    public class CustomersController
        : BaseCrudController<Customer, CustomersController>
    {
        public CustomersController(
            ICustomerRepo customerRepo, IAppLogging<CustomersController> logger)
            : base(customerRepo, logger)
        {
        }
    }
}

// MakesController.cs
using AutoLot.Api.Controllers.Base;
using AutoLot.Models.Entities;
using Microsoft.AspNetCore.Mvc;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.Logging;
namespace AutoLot.Api.Controllers
{
    [Route("api/[controller]")]
    public class MakesController : BaseCrudController<Make, MakesController>
    {
```

```

public MakesController(IMakeRepo makeRepo,
    IAppLogging<MakesController> logger)
    : base(makeRepo, logger)
{
}
}

// OrdersController.cs
using AutoLot.Api.Controllers.Base;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Models.Entities;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Mvc;
namespace AutoLot.Api.Controllers
{
    [Route("api/[controller]")]
    public class OrdersController
        : BaseCrudController<Order, OrdersController>
    {
        public OrdersController(IOrderRepo orderRepo,
            IAppLogging<OrdersController> logger)
            : base(orderRepo, logger)
        {
        }
    }
}

```

Итак, все контроллеры готовы и вы можете с помощью пользовательского интерфейса Swagger протестировать полную функциональность. Если вы собираетесь добавлять/обновлять/удалять записи, тогда измените значение RebuildDataBase на true в файле appsettings.development.json:

```

{
    ...
    "RebuildDataBase": true,
    ...
}

```

Фильтры исключений

Когда в приложении Web API возникает исключение, никакая страница со сведениями об ошибке не отображается, т.к. пользователем обычно является другое приложение, а не человек. Информация об ошибке должна быть отправлена в формате JSON наряду с кодом состояния HTTP. Как обсуждалось в главе 29, инфраструктура ASP.NET Core позволяет создавать фильтры, которые запускаются при появлении необработанных исключений. Фильтры можно применять глобально, на уровне контроллера или на уровне действия. Для текущего приложения вы постройте фильтр исключений для отправки данных JSON (вместе с кодом HTTP 500) и включения трассировки стека, если сайт функционирует в режиме отладки.

На заметку! Фильтры — крайне мощное средство ASP.NET Core. В этой главе вы ознакомитесь только с фильтрами исключений, но с их помощью можно создавать очень многое, что значительно экономит время при построении приложений ASP.NET Core. Полную информацию о фильтрах ищите в документации по ссылке <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/filters>.

Создание специального фильтра исключений

Создайте новый каталог под наименованием *Filters* и добавьте в него новый файл класса по имени *CustomExceptionFilterAttribute.cs*. Приведите операторы *using* к следующему виду:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Hosting;
```

Сделайте класс открытым и унаследованным от *ExceptionFilterAttribute*. Переопределите метод *OnException()*, как показано ниже:

```
namespace AutoLot.Api.Filters
{
    public class CustomExceptionFilterAttribute
        : ExceptionFilterAttribute
    {
        public override void OnException(ExceptionContext context)
        {
        }
    }
}
```

В отличие от большинства фильтров в ASP.NET Core, которые имеют обработчик событий “перед” и “после”, фильтры исключений располагают только одним обработчиком: *OnException()* (или *OnExceptionAsync()*). Обработчик принимает один параметр, *ExceptionContext*, который предоставляет доступ к *ActionContext*, а также к генерированному исключению.

Кроме того, фильтры принимают участие во внедрении зависимостей, позволяя получить доступ в коде к любому элементу внутри контейнера. В рассматриваемом примере вам необходим экземпляр реализации *IWebHostEnvironment*, внедренный в фильтр, который будет использоваться для выяснения среды времени выполнения. Если средой является *Development*, тогда ответ должен также включать трассировку стека. Добавьте переменную уровня класса для хранения экземпляра реализации *IWebHostEnvironment* и конструктор:

```
private readonly IWebHostEnvironment _hostEnvironment;
public CustomExceptionFilterAttribute(
    IWebHostEnvironment hostEnvironment)
{
    _hostEnvironment = hostEnvironment;
}
```

Код в обработчике *OnException()* проверяет тип генерированного исключения и строит соответствующий ответ. В случае среды *Development* в сообщение ответа включается трассировка стека. Затем создается динамический объект, который содержит значения для отправки вызывающему запросу, и возвращается в *IActionResult*. Вот модифицированный код метода:

```
public override void OnException(ExceptionContext context)
{
    var ex = context.Exception;
```

```

string stackTrace = _hostEnvironment.IsDevelopment()
    ? context.Exception.StackTrace :
string.Empty;
string message = ex.Message;
string error;
IActionResult ActionResult;
switch (ex)
{
    case DbUpdateConcurrencyException ce:
        // Возвращается код HTTP 400.
        error = "Concurrency Issue.";
        ActionResult = new BadRequestObjectResult(
            new {Error = error, Message = message, StackTrace = stackTrace});
        break;
    default:
        error = "General Error.";
        ActionResult = new ObjectResult(
            new {Error = error, Message = message, StackTrace = stackTrace})
        {
            StatusCode = 500
        };
        break;
}
// context.ExceptionHandled = true; // Если убрать здесь комментарий,
//                                // то исключение поглощается.
context.Result = ActionResult;
}

```

Если вы хотите, чтобы фильтр исключений поглотил исключение и установил код состояния в 200 (скажем, для регистрации ошибки в журнале, не возвращая ее клиенту), тогда поместите следующую строку перед установкой `Result` (в предыдущем примере кода просто уберите комментарий):

```
context.ExceptionHandled = true;
```

Добавление фильтров в конвейер обработки

Фильтры можно применять к методам действий, контроллерам или глобально к приложению. Код “перед” фильтров выполняется снаружи вовнутрь (глобальный, контроллер, метод действия), в то время как код “после” фильтров выполняется изнутри наружу (метод действия, контроллер, глобальный).

На уровне приложения фильтры добавляются в методе `ConfigureServices()` класса `Startup`. Откройте файл класса `Startup.cs` и поместите в начало файла следующий оператор `using`:

```
using AutoLot.Api.Filters;
```

Модифицируйте метод `AddControllers()`, добавив специальный фильтр:

```

services
    .AddControllers(config => config.Filters.Add(
        new CustomExceptionFilterAttribute(_env)))
    .AddJsonOptions(options =>
{
    options.JsonSerializerOptions.PropertyNamingPolicy = null;
}

```

```

        options.JsonSerializerOptions.WriteIndented = true;
    })
.ConfigureApiBehaviorOptions(options =>
{
...
});

```

Тестирование фильтра исключений

Чтобы протестировать фильтр исключений, откройте файл WeatherForecastController.cs и обновите метод действия Get() показанным ниже кодом:

```

[HttpGet]
public IEnumerable<WeatherForecast> Get()
{
    _logger.LogAppWarning("This is a test");
    throw new Exception("Test Exception");
...
}

```

Запустите приложение и испытайте метод с использованием Swagger. Результаты, отображенные в пользовательском интерфейсе Swagger должны соответствовать следующему выводу (трассировка стека приведена с сокращениями):

```
{
  "Error": "General Error.",
  "Message": "Test Exception",
  "StackTrace":
    " at AutoLot.Api.Controllers.WeatherForecastController.Get() in
D:\\\\Projects\\\\Books\\\\csharp9-wf\\\\Code\\\\New\\\\Chapter_30
\\\\AutoLot.Api\\\\Controllers\\\\WeatherForecastController.cs:line 31\\r\\n "
}
```

Добавление поддержки запросов между источниками

Приложения API должны иметь политики, которые разрешают или запрещают взаимодействовать с ними клиентам, обращающимся из другого сервера. Такие типы запросов называются *запросами между источниками* (cross-origin requests — CORS). Хотя в этом нет необходимости при работе локально на своей машине для всего мира ASP.NET Core, поддержка CORS нужна фреймворкам JavaScript, которые желают взаимодействовать с вашим приложением API, даже когда они все вместе функционируют локально.

На заметку! Дополнительные сведения о поддержке CORS ищите в документации по ссылке <https://docs.microsoft.com/ru-ru/aspnet/core/security/cors>.

Создание политики CORS

Инфраструктура ASP.NET Core располагает развитой поддержкой конфигурирования CORS, включая методы для разрешения/запрещения заголовков, методов, источников, учетных данных и многое другое. В этом примере все будет оставлено максимально открытым.

Конфигурирование начинается с создания политики CORS и добавления ее в коллекцию служб. Политика имеет имя (оно будет использоваться в методе `Configure()`), за которым следуют правила. Далее будет создана политика по имени `AllowAll`, разрешающая все. Добавьте в метод `ConfigureServices()` класса `Startup` следующий код:

```
services.AddCors(options =>
{
    options.AddPolicy("AllowAll", builder =>
    {
        builder
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowAnyOrigin();
    });
});
```

Добавление политики CORS в конвейер обработки HTTP

Наконец, политику CORS необходимо добавить в конвейер обработки HTTP. Поместите между вызовами `app.UseRouting()` и `app.UseEndpoints()` в методе `Configure()` класса `Startup` показанную ниже строку (выделенную полужирным):

```
public void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env,
    ApplicationDbContext context)
{
    ...
    // Включить маршрутизацию.
    app.UseRouting();

    // Добавить политику CORS.
app.UseCors("AllowAll");

    // Включить проверки авторизации.
    app.UseAuthorization();
    ...
}
```

Резюме

В главе вы продолжили изучение ASP.NET Core. Сначала вы узнали о возвращении данных JSON из методов действий, после чего взглянули на атрибут `ApiController` и его влияние на контроллеры API. Затем вы обновили общую реализацию `Swashbuckle`, чтобы включить XML-документацию приложения и информацию из атрибутов методов действий.

Далее был построен базовый контроллер, содержащий большинство функциональности приложения. После этого в проект были добавлены производные контроллеры, специфичные для сущностей. В заключение был добавлен фильтр исключений уровня приложения и поддержка запросов между источниками.

В следующей главе вы завершите построение веб-приложения ASP.NET Core, т.е. `AutoLot.Mvc`.

ГЛАВА 31

Создание приложений MVC с помощью ASP.NET Core

В главе 29 была заложена основа ASP.NET Core, а в главе 30 вы построили службу REST. В этой главе вы будете создавать веб-приложение с использованием паттерна MVC. Все начинается с помещения “V” обратно в “MVC”.

На заметку! Исходный код, рассматриваемый в этой главе, находится в папке `Chapter_31` внутри хранилища GitHub для настоящей книги. Вы также можете продолжить работу с решением, начатым в главе 29 и обновленным в главе 30.

Введение в представления ASP.NET Core

При построении служб ASP.NET Core были задействованы только части “М” (модели) и “С” (контроллеры) паттерна MVC. Пользовательский интерфейс создается с применением части “V”, т.е. представлений паттерна MVC. Представления строятся с использованием кода HTML, JavaScript, CSS и Razor. Они необязательно имеют страницу базовой компоновки и визуализируются из метода действия контроллера или компонента представления. Если вы имели дело с классической инфраструктурой ASP.NET MVC, то все должно выглядеть знакомым.

Экземпляры класса `ViewResult` и методы действий

Как кратко упоминалось в главе 29, объекты результатов `ViewResult` и `PartialView` являются экземплярами класса `ActionResult`, которые возвращаются из методов действий с применением вспомогательных методов класса `Controller`. Класс `PartialViewResult` спроектирован для визуализации внутри другого представления и не использует страницу компоновки, тогда как класс `ViewResult` обычно визуализируется в сочетании со страницей компоновки.

По соглашению, принятому в ASP.NET Core (что было и в ASP.NET MVC), экземпляр `View` или `PartialView` визуализирует файл `*.cshtml` с таким же именем, как у метода. Представление должно находиться либо в каталоге с именем контроллера (без суффикса `Controller`), либо в каталоге `Shared` (оба расположены внутри родительского каталога `Views`).

Например, следующий код будет визуализировать представление SampleAction.cshtml, находящееся в каталоге Views\Sample или Views\Shared:

```
[Route("[controller]/[action]")]
public class SampleController: Controller
{
    public ActionResult SampleAction()
    {
        return View();
    }
}
```

На заметку! Первым производится поиск в каталоге с именем контроллера. Если представление там не обнаружено, то поиск выполняется в каталоге Shared. Если оно по-прежнему не найдено, тогда генерируется исключение.

Чтобы визуализировать представление с именем, которое отличается от имени метода действия, передавайте имя файла (без расширения cshtml). Показанный ниже код будет визуализировать представление CustomViewName.cshtml:

```
public ActionResult SampleAction()
{
    return View("CustomViewName");
}
```

Последние две перегруженные версии предназначены для передачи объекта данных, который становится моделью для представления. В первом примере применяется стандартное имя представления, а во втором указывается другое имя представления:

```
public ActionResult SampleAction()
{
    var sampleModel = new SampleActionViewModel();
    return View(sampleModel);
}
public ActionResult SampleAction()
{
    var sampleModel = new SampleActionViewModel();
    return View("CustomViewName", sampleModel);
}
```

В следующем разделе подробно рассматривается механизм визуализации Razor с использованием представления, которое визуализируется из метода действия по имени RazorSyntax() класса HomeController. Метод действия будет получать запись Car из экземпляра класса CarRepo, внедряемого в метод, и передавать экземпляр Car в качестве модели представлению.

Откройте HomeController в каталоге Controllers приложения AutoLot.Mvc и добавьте следующий оператор using:

```
using AutoLot.Dal.Repos.Interfaces;
```

Затем добавьте в контроллер метод RazorSyntax():

```
[HttpGet]
public IActionResult RazorSyntax([FromServices] ICarRepo carRepo)
{
    var car = carRepo.Find(1);
    return View(car);
}
```

Метод действия декорируется атрибутом `HTTPGet` с целью установки этого метода в качестве конечной точки приложения для `/Home/RazorSyntax` при условии, что поступивший запрос является HTTP-запросом GET. Атрибут `FromServices` на параметре `ICarRepo` информирует ASP.NET Core о том, что параметр не должен привязываться к каким-либо входящим данным, а взамен метод получает экземпляр реализации `ICarRepo` из контейнера DI (`dependency injection` — внедрение зависимостей). Метод получает экземпляр `Car` и возвращает экземпляр `ViewResult` с применением метода `View()`. Поскольку имя представления не было указано, ASP.NET Core будет искать представление с именем `RazorSyntax.cshtml` в каталоге `Views\Home` или `Views\Shared`. Если ни в одном местоположении представление не найдено, тогда клиенту (браузеру) возвратится исключение.

Запустите приложение и перейдите в браузере по ссылке <https://localhost:5001/Home/RazorSyntax> (в случае использования Visual Studio и IIS вам понадобится изменить номер порта). Так как в проекте отсутствует представление, которое может удовлетворить запрос, в браузер возвращается исключение. Вспомните из главы 29, что внутри метода `Configure()` класса `Startup` в конвейер HTTP добавляется вызов `UseDeveloperExceptionPage()`, если средой является `Development`. Результаты работы этого метода показаны на рис. 31.1.

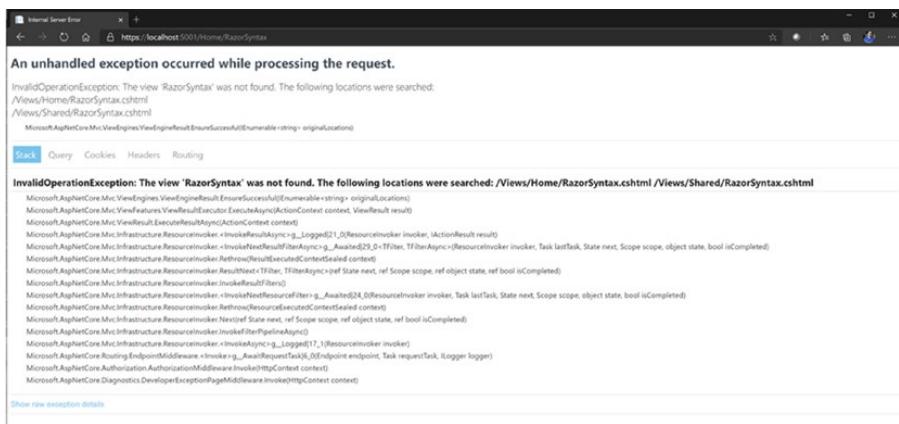


Рис. 31.1. Отображение сообщения об ошибке с помощью страницы исключений для разработчиков

Страница исключений для разработчиков предоставляет обширную информацию для отладки приложения, в числе которой низкоуровневые детали исключения, укомплектованные трассировкой стека. Теперь закомментируйте приведенную ниже строку в методе `Configure()` и замените ее "стандартным" обработчиком ошибок:

```
if (env.IsDevelopment())
{
    // app.UseDeveloperExceptionPage();
    app.UseExceptionHandler("/Home/Error");
    ...
}
```

Снова запустив приложение и перейдя по ссылке `http://localhost:5001/Home/RazorSyntax`, вы увидите стандартную страницу ошибок, которая показана на рис. 31.2.

AutoLot.Mvc Home Privacy

Error.

An error occurred while processing your request.

Request ID: 00-77463896394b784297fb9370c97e395-bf50d9edf7d11d42-00

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The **Development** environment shouldn't be enabled for **deployed applications**. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

Рис. 31.2. Отображение сообщения об ошибке с помощью стандартной страницы ошибок

На заметку! Во всех примерах URL в этой главе применяется веб-сервер Kestrel и порт 5001.

Если вы имеете дело с Visual Studio и веб-сервером IIS Express, тогда используйте URL из профиля для IIS в файле `launchsettings.json`.

Стандартный обработчик ошибок выполняет перенаправление ошибок методу `OnActionExecuting` класса `ErrorController`. Не забудьте восстановить применение страницы исключений для разработчиков в методе `Configure()`:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    ...
}
```

Дополнительные сведения о настройке обработки ошибок и доступных вариантах ищите в документации по ссылке <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/error-handling>.

Механизм визуализации и синтаксис Razor

Механизм визуализации Razor задумывался как усовершенствование механизма визуализации Web Forms и использует Razor в качестве основного языка. Razor — это код серверной стороны, который встраивается в представление, базируется на C# и избавляет от многих неудобств, присущих механизму визуализации Web Forms. Встраивание Razor в HTML и CSS приводит к тому, что код становится намного чище и лучше для восприятия, чем в случае, когда применяется синтаксис механизма визуализации Web Forms.

Первым делом добавьте новое представление, щелкнув правой кнопкой мыши на имени каталога Views\Home в проекте AutoLot.Mvc и выбрав в контекстном меню пункт Add⇒New Item (Добавить⇒Новый элемент). В открывшемся диалоговом окне Add New Item — AutoLot.Mvc (Добавить новый элемент — AutoLot.Mvc) выберите шаблон Razor View — Empty (Представление Razor — Пустое) и назначьте представлению имя RazorSyntax.cshtml.

На заметку! Контекстное меню, открывшееся в результате щелчка правой кнопкой мыши на Views\Home, содержит также пункт Add⇒View (Добавить⇒Представление). Тем не менее, его выбор приводит к переходу в то же самое диалоговое окно Add New Item.

Представления Razor, как правило, строго типизированы с использованием директивы @model (обратите внимание на букву *m* в нижнем регистре). Измените тип нового представления на сущность Car, добавив в начало файла представления такой код:

```
@model AutoLot.Models.Entities.Car
```

Поместите в верхнюю часть страницы дескриптор <h1>. Он не имеет ничего общего с Razor, а просто добавляет заголовок к странице:

```
<h1>Razor Syntax</h1>
```

Блоки операторов Razor открываются с помощью символа @ и являются либо самостоятельными операторами (вроде foreach), либо заключаются в фигурные скобки, как демонстрируется в следующих примерах:

```
@for (var i = 0; i < 15; i++)
{
    // Делать что-то.
}
@{
    // Блок кода.
    var foo = "Foo";
    var bar = "Bar";
    var htmlString = "<ul><li>one</li><li>two</li></ul>";
}
```

Чтобы вывести значение переменной в представление, просто укажите символ @ с именем переменной, что эквивалентно вызову Response.Write(). Как видите, при выводе напрямую в браузер после оператора нет точки с запятой:

```
@foo
<br />
@htmlString
<br />
@foo.@bar
<br />
```

В предыдущем примере две переменные комбинируются посредством точки между ними (@foo.@bar). Это не обычная “точечная” запись в языке C#, предназначенная для навигации по цепочке свойств. Здесь просто значения двух переменных выводятся в поток ответа с физической точкой между ними. Если вас интересует “точечная” запись в отношении переменной, тогда примените @ к переменной и записывайте свой код стандартным образом:

```
@foo.ToUpper()
```

Если вы хотите вывести низкоуровневую HTML-разметку, тогда используйте так называемые *вспомогательные функции HTML* (HTML helper), которые встроены в механизм визуализации Razor. Следующая строка выводит низкоуровневую HTML-разметку:

```
@Html.Raw(htmlString)
<hr />
```

В блоках кода можно смешивать разметку и код. Строки, начинающиеся с разметки, интерпретируются как HTML, а остальные строки — как код. Если строка начинается с текста, который не является кодом, вы должны применять указатель содержимого (@:) или указатель блока содержимого (<text></text>). Обратите внимание, что строки могут меняться с одного вида на другой и наоборот. Ниже приведен пример:

```
@{
    @:Straight Text
    <div>Value:@Model.Id</div>
    <text>
        Lines without HTML tag
    </text>
    <br />
}
```

При желании отменить символ @ используйте удвоенный @. Кроме того, механизм Razor достаточно интеллектуален, чтобы обрабатывать адреса электронной почты, поэтому отменять символ @ в них не нужно. Если необходимо заставить Razor трактовать символ @ подобно маркеру Razor, тогда добавьте круглые скобки:

```
foo@foo.com
<br />
@@foo
<br />
test@foo
<br/>
test@(foo)
<br />
```

Предыдущий код выводит foo@foo.com, @foo, test@foo и testFoo.

Комментарии Razor открываются с помощью @* и закрываются посредством *@:

```
@*
    Multiline Comments
    Hi.
*@
```

В Razor также поддерживаются внутристрочные функции. Например, следующая функция сортирует список строк:

```
@functions {
    public static IList<string> SortList(IList<string> strings) {
        var list = from s in strings orderby s select s;
        return list.ToList();
    }
}
```

Приведенный далее код создает список строк, сортирует их с применением функции `SortList()` и выводит отсортированный список в браузер:

```
@{
    var myList = new List<string> {"C", "A", "Z", "F"};
    var sortedList = SortList(myList);
}
@foreach (string s in sortedList)
{
    @s@:&nbsp;
}
<hr/>
```

Вот еще один пример, где создается делегат, который можно использовать, чтобы установить для строки полужирное начертание:

```
@{
    Func<dynamic, object> b = @<strong>@item</strong>;
}
This will be bold: @b("Foo")
```

Кроме того, Razor содержит вспомогательные методы HTML, которые предстаются инфраструктурой ASP.NET Core, например, `DisplayForModel()` и `EditorForModel()`. Первый применяет рефлексию к модели представления для отображения на веб-странице. Второй тоже использует рефлексию, чтобы создать HTML-разметку для формы редактирования (имейте в виду, что он не поставляет дескрипторы `Form`, а только разметку для модели). Вспомогательные методы HTML подробно рассматриваются позже в главе.

Наконец, в версии ASP.NET Core появились *вспомогательные функции дескрипторов* (`tag helper`), которые объединяют разметку и код; они будут обсуждаться далее в главе.

Представления

Представления — это специальные файлы кода с расширением `cshtml`, содержащие сочетание разметки HTML, стилей CSS, кода JavaScript и кода Razor.

Каталог Views

Внутри каталога `Views` хранятся представления в проектах ASP.NET Core, использующих паттерн MVC. В самом каталоге `Views` находятся два файла: `_ViewStart.cshtml` и `_ViewImports.cshtml`.

Код в файле `_ViewStart.cshtml` выполняется перед визуализацией любого другого представления (за исключением частичных представлений и компоновок). Файл `_ViewStart.cshtml` обычно применяется с целью установки стандартной компоновки для представлений, в которых она не указана. Компоновки подробно рассматриваются в разделе “Компоновки” позже в главе. Вот как выглядит содержимое файла `_ViewStart.cshtml`:

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

Файл `_ViewImports.cshtml` служит для импортирования совместно используемых директив, таких как операторы `using`. Содержимое применяется ко всем пред-

ставлениям в том же каталоге или подкаталоге, где находится файл `_ViewImports.cshtml`. Добавьте оператор `using` для `AutoLot.Models.Entities`:

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

Строка `@addTagHelper` будет раскрыта вместе со вспомогательными функциями дескрипторов.

На заметку! А для чего служит ведущий символ подчеркивания в `_ViewStart.html`, `_ViewImports.cshtml` и `_Layout.cshtml`? Механизм визуализации Razor изначально создавался для платформы WebMatrix, где не разрешалось напрямую визуализировать файлы, имена которых начинались с символа подчеркивания. Все ключевые файлы (вроде компоновки и конфигурации) имеют имена, начинающиеся с символа подчеркивания. Это не соглашение MVC, поскольку здесь отсутствует проблема, которая была в WebMatrix, но наследие символа подчеркивания продолжает существовать.

Как упоминалось ранее, каждый контроллер получает собственный каталог внутри каталога `Views`, в котором хранятся его специфичные представления. Имя такого каталога совпадает с именем контроллера (без суффикса `Controller`). Скажем, в каталоге `Views\Cars` содержатся все представления для `CarsController`. Представления обычно именуются согласно методам действий, которые их визуализируют, хотя их имена можно изменять, как уже было показано.

Каталог `Shared`

Внутри каталога `Views` есть специальный каталог по имени `Shared`, в котором хранятся представления, доступные всем контроллерам и действиям. Как уже упоминалось, если запрошенный файл представления не удалось найти в каталоге, специфичном для контроллера, тогда поиск производится в каталоге `Shared`.

Каталог `DisplayTemplates`

В каталоге `DisplayTemplates` хранятся специальные шаблоны, которые управляют визуализацией типов, а также содействуют многократному использованию кода и согласованности отображения. Когда вызываются методы `DisplayFor()` / `DisplayForModel()`, механизм визуализации Razor ищет шаблон, имя которого совпадает с именем визуализируемого типа, например, `Car.cshtml` для класса `Car`. Если специальный шаблон найти не удалось, тогда разметка визуализируется с применением рефлексии. Поиск начинается с каталога `Views\{CurrentControllerName}\DisplayTemplates` и в случае неудачи продолжается в каталоге `Views\Shared\DisplayTemplates`. Методы `DisplayFor()` / `DisplayForModel()` принимают необязательный параметр, указывающий имя шаблона.

Шаблон отображения `DateTime`

Создайте внутри каталога `Views\Shared` новый каталог под названием `DisplayTemplates` и добавьте в него новое представление по имени `DateTime.cshtml`. Удалите сгенерированный код вместе с комментариями и замените его следующим кодом:

```
@model DateTime?
@if (Model == null)
{
    @:Unknown
}
else
{
    if (ViewData.ModelMetadata.IsNullableValueType)
    {
        @:@(Model.Value.ToString("d"))
    }
    else
    {
        @:@(((DateTime)Model).ToString("d"))
    }
}
```

Обратите внимание, что в директиве `@model`, строго типизирующей представление, используется буква `m` нижнего регистра. При ссылке на присвоенное значение модели в Razor применяется буква `M` верхнего регистра. В этом примере определение модели допускает значения `null`. Если переданное представлению значение для модели равно `null`, то шаблон отображает слово `Unknown` (неизвестно). В противном случае шаблон отображает дату в сокращенном формате, используя свойство `Value` допускающего `null` типа или саму модель.

Шаблон отображения Car

Создайте внутри каталога `Views` новый каталог по имени `Cars`, а внутри него — каталог под названием `DisplayTemplates`. Добавьте в каталог `DisplayTemplates` новое представление по имени `Car.cshtml`. Удалите сгенерированный код вместе с комментариями и замените его показанным ниже кодом, который отображает сущность `Car`:

```
@model AutoLot.Models.Entities.Car
<dl class="row">
    <dt class="col-sm-2">
        @Html.DisplayNameFor(model => model.MakeId)
    </dt>
    <dd class="col-sm-10">
        @Html.DisplayFor(model => model.MakeNavigation.Name)
    </dd>
    <dt class="col-sm-2">
        @Html.DisplayNameFor(model => model.Color)
    </dt>
    <dd class="col-sm-10">
        @Html.DisplayFor(model => model.Color)
    </dd>
    <dt class="col-sm-2">
        @Html.DisplayNameFor(model => model.PetName)
    </dt>
    <dd class="col-sm-10">
        @Html.DisplayFor(model => model.PetName)
    </dd>
</dl>
```

Вспомогательная функция HTML под названием `DisplayNameFor()` отображает имя свойства, если только свойство не декорировано или атрибутом `Display(Name = "")`, или атрибутом `DisplayName("")`, и тогда применяется отображаемое значение. Метод `DisplayFor()` отображает значение для свойства модели, указанное в выражении. Обратите внимание, что для получения названия производителя используется навигационное свойство `MakeNavigation`.

Запустив приложение и перейдя на страницу `RazorSyntax`, вы можете быть удивлены тем, что шаблон отображения `Car` не применяется. Причина в том, что шаблон находится в каталоге представления `Cars`, а метод действия `RazorSyntax` и представление вызываются из `HomeController`. Методы действий в `HomeController` будут осуществлять поиск представлений в каталогах `Home` и `Shared` и потому не найдут шаблон отображения `Car`.

Если вы переместите файл `Car.cshtml` в каталог `Shared\DisplayTemplates`, тогда представление `RazorSyntax` будет использовать шаблон отображения `Car`.

Шаблон отображения `CarWithColor`

Шаблон `CarWithColor` похож на шаблон `Car`. Разница в том, что этот шаблон изменяет цвет текста `Color` (Цвет) на основе значения свойства `Color` модели. Добавьте в каталог `Cars\DisplayTemplates` новый шаблон по имени `CarWithColors.cshtml` и приведите разметку к следующему виду:

```
@model Car
<hr />
<div>
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.PetName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.PetName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.MakeNavigation)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.MakeNavigation.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Color)
        </dt>
        <dd class="col-sm-10" style="color:@Model.Color">
            @Html.DisplayFor(model => model.Color)
        </dd>
    </dl>
</div>
```

Чтобы применить шаблон `CarWithColors.cshtml` вместо `Car.cshtml`, вызовите `DisplayForModel()` с именем шаблона (обратите внимание, что правила местоположения по-прежнему актуальны):

```
@Html.DisplayForModel("CarWithColors")
```

Каталог EditorTemplates

Каталог EditorTemplates работает аналогично каталогу DisplayTemplates, но находящиеся в нем шаблоны используются для редактирования.

Шаблон редактирования Car

Создайте внутри каталога Views\Cars новый каталог под названием EditorTemplates и добавьте в него новое представление по имени Car.cshtml. Удалите сгенерированный код вместе с комментариями и замените его показанным ниже кодом, который является разметкой для редактирования сущности Car:

```
@model Car
<div asp-validation-summary="All" class="text-danger"></div>
<div class="form-group">
    <label asp-for="PetName" class="col-form-label"></label>
    <input asp-for="PetName" class="form-control" />
    <span asp-validation-for="PetName" class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="MakeId" class="col-form-label"></label>
    <select asp-for="MakeId" class="form-control" asp-items="ViewBag.MakeId">
        </select>
</div>
<div class="form-group">
    <label asp-for="Color" class="col-form-label"></label>
    <input asp-for="Color" class="form-control"/>
    <span asp-validation-for="Color" class="text-danger"></span>
</div>
```

В шаблоне редактирования задействовано несколько вспомогательных функций дескрипторов (asp-for, asp-items, asp-validation-for и asp-validation-summary), которые рассматриваются позже в главе.

Шаблон редактирования Car вызывается с помощью вспомогательных функций HTML, которые называются EditorFor() и EditorForModel(). Подобно шаблонам отображения упомянутые функции будут искать представление с именем Car.cshtml или с таким же именем, как у метода.

Компоновки

По аналогии с мастер-страницами Web Forms в MVC поддерживаются компоновки, которые совместно используются представлениями, чтобы обеспечить согласованный внешний вид страниц сайта. Перейдите в каталог Views\Shared и откройте файл _Layout.cshtml. Это полноценный HTML-файл с дескрипторами <head> и <body>.

Файл _Layout.cshtml является основой, в которую визуализируются другие представления. Кроме того, поскольку большая часть страницы (такая как разметка для навигации и верхнего и/или нижнего колонтитула) поддерживается страницей компоновки, страницы представлений сохраняются небольшими и простыми. Найдите в файле _Layout.cshtml следующую строку кода Razor:

```
@RenderBody()
```

Эта строка указывает странице компоновки, где визуализировать представление. Теперь перейдите к строке, расположенной прямо перед закрывающим дескрип-

тором </body>, которая создает новый раздел для компоновки и объявляет его необязательным:

```
@await RenderSectionAsync("scripts", required: false)
```

Разделы также могут помечаться как обязательные путем передачи для второго параметра (required) значения true. Вдобавок они могут визуализироваться синхронным образом:

```
@RenderSection("Header", true)
```

Любой код и/или разметка в блоке @section файла представления будет визуализироваться не там, где вызывается @RenderBody(), а в месте определения раздела, присутствующего в компоновке. Например, пусть у вас есть представление со следующей реализацией раздела:

```
@section Scripts {
    <script src "~/lib/jquery-validation/dist/jquery.validate.js">
    </script>
}
```

Код из представления визуализируется в компоновке на месте определения раздела. Если компоновка содержит показанное ниже определение:

```
<script src "~/lib/jquery/dist/jquery.min.js"></script>
<script src "~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src "~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("Scripts", required: false)
```

тогда будет добавлен раздел представления, приводя в результате к отправке браузеру следующей разметки:

```
<script src "~/lib/jquery/dist/jquery.min.js"></script>
<script src "~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src "~/js/site.js" asp-append-version="true"></script>
<script src "~/lib/jquery-validation/dist/jquery.validate.js"></script>
```

В ASP.NET Core появились два новых метода: IgnoreBody() и IgnoreSection(). В случае помещения внутрь компоновки эти методы отменяют визуализацию тела представления или указанного раздела соответственно. Они позволяют включать или отключать функции представления в компоновке на основе условной логики, такой как уровни безопасности.

Указание стандартной компоновки для представлений

Как упоминалось ранее, стандартная страница компоновки определяется в файле _ViewStart.cshtml. Любое представление, где не указана компоновка, будет использовать компоновку, определенную в первом файле _ViewStart.cshtml, который обнаруживается в каталоге представления или выше него в структуре каталогов.

Частичные представления

Частичные представления концептуально похожи на пользовательские элементы управления в Web Forms. Частичные представления удобны для инкапсуляции пользовательского интерфейса, что помогает сократить объем повторяющегося кода и/или разметки. Частичное представление не задействует компоновку и внедряется внутрь другого представления или визуализируется с помощью компонента представления (рассматривается позже в главе).

Обновление компоновки с использованием частичных представлений

Временами файлы могут становиться большими и громоздкими. Один из способов справиться с такой проблемой предусматривает разбиение компоновки на набор специализированных частичных представлений.

Создание частичных представлений

Создайте внутри каталога Shared новый каталог под названием `Partials` и добавьте в него три пустых представления с именами `_Head.cshtml`, `_JavaScriptFiles.cshtml` и `_Menu.cshtml`.

Частичное представление Head

Вырежьте содержимое между дескрипторами `<head></head>` в компоновке и вставьте его в файл `_Head.cshtml`:

```
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>@ ViewData["Title"] - AutoLot.Mvc</title>
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
<link rel="stylesheet" href="~/css/site.css" />
```

Замените разметку, удаленную из файла `_Layout.cshtml`, вызовом для визуализации нового частичного представления:

```
<head>
  <partial name="Partials/_Head"/>
</head>
```

Дескриптор `<partial>` — это еще один пример вспомогательной функции дескриптора. В атрибуте `name` указывается имя частичного представления с путем, начинающимся с текущего каталога представления, которым в данном случае является `Views\Shared`.

Частичное представление Menu

Для частичного представления `Menu` вырежьте всю разметку между дескрипторами `<header></header>` (не `<head></head>`) и вставьте ее в файл `_Menu.cshtml`. Модифицируйте файл `_Layout.cshtml`, чтобы визуализировать частичное представление `Menu`:

```
<header>
  <partial name="Partials/_Menu"/>
</header>
```

Частичное представление JavaScriptFiles

Наконец, вырежьте дескрипторы `<script>` для файлов JavaScript и вставьте их в частичное представление `JavaScriptFiles`. Удостоверьтесь в том, что оставил дескриптор `RenderSection` на своем месте. Вот частичное представление `JavaScriptFiles`:

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
```

Ниже приведена текущая разметка в файле _Layout.cshtml:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <partial name="Partials/_Head" />
</head>
<body>
    <header>
        <partial name="Partials/_Menu" />
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - AutoLot.Mvc - <a asp-area="" asp-controller="Home"
                asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <partial name="Partials/_JavaScriptFiles" />
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

Отправка данных представлениям

Существует несколько способов отправки данных представлению. В случае строго типизированных представлений данные можно отправлять, когда представления визуализируются (либо из метода действия, либо через вспомогательную функцию дескриптора `<partial>`).

Строго типизированные представления и модели представлений

При передаче методу `View()` модели или модели представления значение присваивается свойству `@model` строго типизированного представления (обратите внимание на букву `m` в нижнем регистре):

```
@model IEnumerable<Order>
```

Свойство `@model` устанавливает тип для представления, к которому затем можно получать доступ с использованием Razor-команды `@Model` (обратите внимание на букву `M` в верхнем регистре):

```
@foreach (var item in Model)
{
    // Делать что-то.
}
```

В методе действия `RazorViewSyntax()` демонстрируется представление, получающее данные из этого метода действия:

```
[HttpGet]
public IActionResult RazorSyntax([FromServices] ICarRepo carRepo)
```

```
{
    var car = carRepo.Find(1);
    return View(car);
}
```

Значение модели может быть передано и в <partial>, как показано ниже:

```
<partial name="Partials/_CarListPartial" model="@Model"/>
```

Объекты ViewBag, ViewData и TempData

Объекты ViewBag, ViewData и TempData являются механизмами для отправки представлению данных небольшого объема. В табл. 31.1 описаны три механизма передачи данных из контроллера в представление (помимо свойства Model) либо из контроллера в контроллер.

Таблица 31.1. Дополнительные способы отправки данных представлению

Объект передачи данных	Описание
TempData	Недолговечный объект, который работает только в рамках текущего и следующего запросов. Обычно используется при перенаправлении на другой метод действия
ViewData	Словарь, который позволяет хранить значения в виде пар "имя-значение" (например, ViewData["Title"] = "My Page")
ViewBag	Динамическая оболочка для словаря ViewData (например, ViewBag.Title = "My Page")

И ViewBag, и ViewData указывают на тот же самый объект; они просто предлагают разные способы доступа к данным. Еще раз взгляните на созданный ранее файл _HeadPartial.cshtml (важная строка выделена полужирным):

```
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>@ViewData["Title"] - AutoLot.Mvc</title>
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
<link rel="stylesheet" href="~/css/site.css" />
```

Вы заметите, что в атрибуте <title> для установки значения применяется объект ViewData. Поскольку ViewData — конструкция Razor, она предваряется символом @. Чтобы увидеть результаты, модифицируйте представление RazorSyntax.cshtml следующим образом:

```
@model AutoLot.Models.Entities.Car
{
    ViewData["Title"] = "RazorSyntax";
}
<h1>Razor Syntax</h1>
...
```

Теперь после запуска приложения и перехода по ссылке <https://localhost:5001/Home/RazorSyntax> вы увидите на вкладке браузера заголовок Razor Syntax — AutoLot.Mvc (Синтаксис Razor — AutoLot.Mvc).

Вспомогательные функции дескрипторов

Вспомогательные функции дескрипторов являются новым средством, введенным в версии ASP.NET Core. *Вспомогательная функция дескриптора* (*tag helper*) — это разметка (специальный дескриптор или атрибут в стандартном дескрипторе), представляющий код серверной стороны, который затем помогает сформировать выпускаемую HTML-разметку. Они значительно совершенствуют процесс разработки и улучшают читабельность представлений MVC.

В отличие от вспомогательных функций HTML, которые вызываются как методы Razor, вспомогательные функции дескрипторов представляют собой атрибуты, добавляемые к стандартным HTML-элементам или автономным специальным дескрипторам. В случае использования для разработки среды Visual Studio появляется дополнительное преимущество в виде средства IntelliSense, которое отображает подсказки по встроенным вспомогательным функциям дескрипторов.

Например, показанная ниже вспомогательная функция HTML создает метку для свойства FullName заказчика:

```
@Html.Label("FullName", "Full Name:", new { @class = "customer" })
```

В итоге генерируется следующая HTML-разметка:

```
<label class="customer" for="FullName">Full Name:</label>
```

По всей видимости, синтаксис вспомогательных функций HTML хорошо понятен разработчикам на языке C#, применяющим ASP.NET MVC и Razor. Но его нельзя считать интуитивно понятным, особенно для тех, кто имеет дело с HTML/CSS/JavaScript, но не с языком C#.

Версия в виде вспомогательной функции дескриптора выглядит так:

```
<label class="customer" asp-for="FullName">Full Name:</label>
```

Она производит тот же самый вывод, но вспомогательные функции дескрипторов благодаря своей интеграции с дескрипторами HTML удерживают разработчика «в рамках разметки».

Существует множество встроенных вспомогательных функций дескрипторов, которые предназначены для применения вместо соответствующих им вспомогательных функций HTML. Однако не все вспомогательные функции HTML имеют ассоциированные вспомогательные функции дескрипторов. В табл. 31.2 перечислены самые распространенные вспомогательные функции дескрипторов, соответствующие им вспомогательные функции HTML и доступные атрибуты. Они будут раскрыты более подробно в оставшейся части главы.

Включение вспомогательных функций дескрипторов

Вспомогательные функции дескрипторов потребуется сделать видимыми любому коду, где их желательно использовать. Файл _ViewImports.html из стандартного шаблона уже содержит следующую строку:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Строка делает все вспомогательные функции дескрипторов из сборки Microsoft.AspNetCore.Mvc.TagHelpers (содержащей все встроенные вспомогательные функции дескрипторов) доступными всем представлениям на уровне каталога с файлом _ViewImports.cshtml и ниже него в иерархии каталогов.

Таблица 31.2. Широко используемые встроенные вспомогательные функции дескрипторов

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для формы	Html.BeginForm() Html.BeginRouteForm() Html.AntiForgeryToken()	asp-route — для именованных маршрутов (не может использоваться с атрибутами контроллера или действия) asp-antiforgery — если должна быть добавлена защита от подделки (по умолчанию true) asp-area — имя области asp-controller — имя контроллера asp-action — имя действия asp-route-<ИмяПараметра> — добавляет параметр к маршруту, например, asp-route-id="1" asp-page — имя страницы Razor asp-page-handler — имя обработчика страницы Razor asp-all-route-data — словарь для дополнительных значений маршрута
Вспомогательная функция дескриптора для действия формы (<button> или <input type=image>)	—	asp-route — для именованных маршрутов (не может использоваться с атрибутами контроллера или действия) asp-antiforgery — если должна быть добавлена защита от подделки (по умолчанию true) asp-area — имя области asp-controller — имя контроллера asp-action — имя действия asp-route-<ИмяПараметра> — добавляет параметр к маршруту, например, asp-route-id="1" asp-page — имя страницы Razor asp-page-handler — имя обработчика страницы Razor asp-all-route-data — словарь для дополнительных значений маршрута

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для якоря	Html.ActionLink()	asp-route — для именованных маршрутов (не может использоваться с атрибутами контроллера или действия) asp-area — имя области asp-controller — имя контроллера asp-action — имя действия asp-protocol — HTTP или HTTPS asp-fragment — фрагмент URL asp-host — имя хоста asp-route- <i><ИмяПараметра></i> — добавляет параметр к маршруту, например, asp-route-id="1" asp-page — имя страницы Razor asp-page-handler — имя обработчика страницы Razor asp-all-route-data — словарь для дополнительных значений маршрута
Вспомогательная функция дескриптора для элемента ввода	Html.TextBox() / Html.TextBoxFor() Html.Editor() / Html.EditorFor()	asp-for — свойство модели. Позволяет перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="@локальная Переменная"). Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 и атрибуты type генерируются автоматически
Вспомогательная функция дескриптора для текстовой области	Html.TextAreaFor()	asp-for — свойство модели. Позволяет перемещаться по модели (Customer.Address.Description) и применять выражения (asp-for="@локальная Переменная"). Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 и атрибуты type генерируются автоматически

Продолжение табл. 31.2

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для метки	Html.LabelFor()	asp-for — свойство модели. Позволяет перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="@локальнаяПеременная"). Отображает значение атрибута Display, если он существует, а иначе использует имя свойства
Вспомогательная функция дескриптора для частичного представления	Html.Partial() Html.PartialAsync() Html.RenderPartial() Html.RenderPartialAsync()	name — путь и имя частичного представления for — выражение модели (ModelExpression) в текущей форме должно быть моделью в частичном представлении model — объект, который должен быть моделью в частичном представлении view-data — объект ViewData для частичного представления
Вспомогательная функция дескриптора для элемента выбора	Html.DropDownListFor() Html.ListBoxFor()	asp-for — свойство модели. Позволяет перемещаться по модели (Customer.Address.AddressLine1) и применять выражения (asp-for="[@локальнаяПеременная"]") asp-items — указывает элементы option. Атрибут selected генерируется автоматически. Атрибуты id и name генерируются автоматически. Любые атрибуты data-val из HTML5 генерируются автоматически
Вспомогательная функция дескриптора для сообщения проверки достоверности ()	Html.ValidationMessageFor()	asp-validation-for — свойство модели. Позволяет перемещаться по модели (Customer.Address.AddressLine1) и изменять выражения (asp-for="[@локальнаяПеременная"]"). Добавляет к атрибут data-valmsg-for

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для сводки по проверке достоверности (<div>)	Html.ValidationSummaryFor()	asp-validation-summary — один из вариантов All, ModelOnly или None. Добавляет к <div> атрибут data-valmsg-summary
Вспомогательная функция дескриптора для ссылки	—	<p>asp-append-version — добавляет хеш-значение файла в качестве указателя версии к имени файла (в виде строки запроса) с целью очистки кеша</p> <p>href — адрес для версии источника в сети доставки содержимого</p> <p>asp-fallback-href — запасной файл, который должен использоваться, если основной не доступен; обычно применяется с источниками в сетях доставки содержимого</p> <p>asp-fallback-href-include — универсализированный список файлов для включения при переходе на запасной вариант</p> <p>asp-fallback-href-exclude — универсализированный список файлов для исключения при переходе на запасной вариант</p> <p>asp-fallback-test-* — свойства, которые должны использоваться при проверке на предмет перехода на запасной вариант. Включают class, property и value</p> <p>asp-href-include — универсализированный шаблон для включения файлов</p> <p>asp-href-exclude — универсализированный шаблон для исключения файлов</p>

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для сценария	–	<p>asp-append-version — добавляет хеш-значение файла в качестве указателя версии к имени файла (в виде строки запроса) с целью очистки кеша</p> <p>src — адрес для версии источника в сети доставки содержимого</p> <p>asp-fallback-src — запасной файл, который должен использоваться, если основной не доступен; обычно применяется с источниками в сетях доставки содержимого</p> <p>asp-fallback-src-include — универсализированный список файлов для включения при переходе на запасной вариант</p> <p>asp-fallback-src-exclude — универсализированный список файлов для исключения при переходе на запасной вариант</p> <p>asp-fallback-test — сценарный метод для использования при проверке на предмет перехода на запасной вариант</p> <p>asp-src-include — универсализированный шаблон для включения файлов</p> <p>asp-src-exclude — универсализированный шаблон для исключения файлов</p>
Вспомогательная функция дескриптора для изображения	–	<p>asp-append-version — добавляет хеш-значение файла в качестве указателя версии к имени файла (в виде строки запроса) с целью очистки кеша</p>

Вспомогательная функция дескриптора	Вспомогательная функция HTML	Доступные атрибуты
Вспомогательная функция дескриптора для среды	–	<p>names — одиночное имя размещающей среды или разделенный запятыми список имен для запуска визуализации содержимого (регистр символов игнорируется)</p> <p>include — одиночное имя размещающей среды или разделенный запятыми список имен для включения визуализацию содержимого (регистр символов игнорируется)</p> <p>exclude — одиночное имя размещающей среды или разделенный запятыми список имен для исключения из визуализации содержимого (регистр символов игнорируется)</p>

Вспомогательная функция дескриптора для формы

Вспомогательная функция дескриптора для формы (`<form>`) заменяет вспомогательные функции HTML с именами `Html.BeginForm()` и `Html.BeginRouteForm()`. Скажем, чтобы создать форму, которая отправляет версию действия `Edit` для HTTP-метода `POST` контроллера `CarsController` с одним параметром `{Id}`, потребуется следующий код и разметка:

```
<form method="post" asp-controller="Cars" asp-action="Edit"
      asp-route-id="@Model.Id" >
    <!-- Для краткости не показано -->
</form>
```

С точки зрения строгой HTML-разметки дескриптор `<form>` будет работать без атрибутов вспомогательной функции дескриптора для формы. Если атрибуты отсутствуют, тогда это просто обычная HTML-форма, к которой понадобится вручную добавить маркер защиты от подделки. Тем не менее, после добавления одного из атрибутов `asp-*` в форме добавляется и маркер защиты от подделки, который можно отключить, добавив к дескриптору `<form>` атрибут `asp-antiforgery="false"`. Маркер защиты от подделки рассматривается позже в главе.

Форма создания для сущности `Car`

Форма создания для сущности `Car` отправляется методу действия `Create()` класса `CarsController`. Добавьте в каталог `Views\Cars` новое пустое представление Razor по имени `Create.cshtml` со следующим содержимым:

```
@model Car
 @{
    ViewData["Title"] = "Create";
}
```

```

<h1>Create a New Car</h1>
<hr/>
<div class="row">
  <div class="col-md-4">
    <form asp-controller="Cars" asp-action="Create">
      </form>
    </div>
  </div>

```

Хотя представление не полное, его достаточно для демонстрации того, что было раскрыто до сих пор, а также вспомогательной функции дескриптора для формы. Первая строка строго типизирует представление сущностным классом `Car`. Блок кода Razor устанавливает специфичный к представлению заголовок для страницы. HTML-дескриптор `<form>` имеет атрибуты `asp-controller` и `asp-action`, которые выполняются на серверной стороне для формирования дескриптора, а также добавления маркера защиты от подделки. Чтобы визуализировать это представление, добавьте в каталог `Controllers` новый контроллер по имени `CarsController`. Модифицируйте код, как показано ниже (позже в главе он будет обновлен):

```

using Microsoft.AspNetCore.Mvc;
namespace AutoLot.Mvc.Controllers
{
    [Route("[controller]/[action]")]
    public class CarsController : Controller
    {
        public IActionResult Create()
        {
            return View();
        }
    }
}

```

Теперь запустите приложение и перейдите по ссылке `http://localhost:5001/Cars/Create`. Инспектирование источника покажет, что форма имеет атрибут действия (`action`), основанный на `asp-controller` и `asp-action`, метод (`method`), установленный в `post`, и добавленный скрытый элемент `<input>` с именем `_RequestVerificationToken`:

```

<form action="/Cars/Create" method="post">
  <input name="_RequestVerificationToken" type="hidden"
  value="CfDJ8Hqg5HsrvCtOkkLRHY4ukxwvix0vkQ3vOvezvtJWd10P5lwbI5-FFWXh8KCF
  Z07eKxveCuK8NRJywj8Jz23pP2nV37fIGqqcITRyISGgq7tRYZDuPv8NMIZz2nCWRiDbxOv
  1kg61DTDW9BrJxr8H63Y">
</form>

```

Далее в главе представление `Create` будет неоднократно обновляться.

Вспомогательная функция дескриптора для действия формы

Вспомогательная функция дескриптора для действия формы используется в элементах кнопок и изображений с целью изменения действия содержащей их формы. Например, следующая кнопка, добавленная к форме редактирования, вызовет передачу запроса POST конечной точке `Create`:

```
<button type="submit" asp-action="Create">Index</button>
```

Вспомогательная функция дескриптора для якоря

Вспомогательная функция дескриптора для якоря (`<a>`) заменяет вспомогательную функцию HTML с именем `Html.ActionLink()`. Скажем, чтобы создать ссылку на представление `RazorSyntax`, применяйте такой код:

```
<a class="nav-link text-dark" asp-area="" asp-controller="Home"
    asp-action="RazorSyntax">
    Razor Syntax
</a>
```

Для добавления страницы синтаксиса `Razor` в меню модифицируйте `_Menu.cshtml`, как показано ниже, добавив новый элемент меню между элементами `Home` (Домой) и `Privacy` (Секретность) (дескрипторы ``, окружающие дескрипторы якорей, предназначены для меню `Bootstrap`):

```
...
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home"
        asp-action="Index">Home</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home"
        asp-action="RazorSyntax">Razor Syntax</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home"
        asp-action="Privacy">Privacy</a>
</li>
```

Вспомогательная функция дескриптора для элемента ввода

Вспомогательная функция дескриптора для элемента ввода (`<input>`) является одной из наиболее универсальных. В дополнение к автоматической генерации атрибутов `id` и `name` стандарта HTML, а также любых атрибутов `data-val` стандарта HTML5, вспомогательная функция дескриптора строит надлежащую HTML-разметку, основываясь на типе данных целевого свойства. В табл. 31.3 перечислены типы HTML, которые создаются на базе типов .NET Core свойств.

Таблица 31.3. Типы HTML, генерируемые из типов .NET с использованием вспомогательной функции дескриптора для элемента ввода

Тип .NET	Генерируемый тип HTML
<code>Bool</code>	<code>type="checkbox"</code>
<code>String</code>	<code>type="text"</code>
<code>DateTime</code>	<code>type="datetime"</code>
<code>Byte, Int, Single, Double</code>	<code>type="number"</code>

Кроме того, вспомогательная функция дескриптора для элемента ввода добавит атрибуты `type` из HTML5, основываясь на аннотациях данных. В табл. 31.4 перечислены некоторые распространенные аннотации и генерируемые атрибуты `type` из HTML5.

Таблица 31.4. Атрибуты type из HTML5, генерируемые на основе аннотаций данных

Аннотация данных .NET	Генерируемый атрибут type в HTML5
EmailAddress	type="email"
Url	type="url"
HiddenInput	type="hidden"
Phone	type="tel"
DataType(DataType.Password)	type="password"
DataType(DataType.Date)	type="date"
DataType(DataType.Time)	type="time"

Шаблон редактирования Car.cshtml содержит дескрипторы <input> для свойств PetName и Color. В качестве напоминания ниже приведены только эти дескрипторы:

```
<input asp-for="PetName" class="form-control" />
<input asp-for="Color" class="form-control"/>
```

Вспомогательная функция дескриптора для элемента ввода добавляет к визуализируемому дескриптору атрибуты name и id, существующее значение для свойства (если оно есть) и атрибуты проверки достоверности HTML5. Оба поля являются обязательными и имеют ограничение на длину строки в 50 символов. Вот визуализированная разметка для указанных двух свойств:

```
<input class="form-control" type="text" data-val="true"
       data-val-length="The field Pet Name must be a string with a
maximum length of 50." data-val-length-max="50"
       data-val-required="The Pet Name field is required."
       id="PetName" maxlength="50" name="PetName" value="Zippy">
<input class="form-control valid" type="text" data-val="true"
       data-val-length="The field Color must be a string with a
maximum length of 50." data-val-length-max="50"
       data-val-required="The Color field is required."
       id="Color" maxlength="50" name="Color" value="Black"
       aria-describedby="Color-error" aria-invalid="false">
```

Вспомогательная функция дескриптора для текстовой области

Вспомогательная функция дескриптора для текстовой области (<textarea>) автоматически добавляет атрибуты id и name и любые атрибуты проверки достоверности HTML5, определенные для свойства. Например, следующая строка создает дескриптор <textarea> для свойства Description:

```
<textarea asp-for="Description"></textarea>
```

Вспомогательная функция дескриптора для элемента выбора

Вспомогательная функция дескриптора для элемента выбора (<select>) создает дескрипторы ввода с выбором из свойства модели и коллекции. Как и в других вспомогательных функциях дескрипторов для элементов ввода, к разметке автоматически добавляются атрибуты id и name, а также любые атрибуты data-val из HTML5. Если значение свойства модели совпадает с одним из значений в списке, тогда для этого варианта в разметку добавляется атрибут selected.

Например, пусть имеется модель со свойством по имени Country и список SelectList по имени Countries с таким определением:

```
public List<SelectListItem> Countries { get; }
    = new List<SelectListItem>
{
    new SelectListItem { Value = "MX", Text = "Mexico" },
    new SelectListItem { Value = "CA", Text = "Canada" },
    new SelectListItem { Value = "US", Text = "USA" },
};
```

Следующая разметка будет визуализировать дескриптор <select> с надлежащими дескрипторами <option>:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Если значением свойства Country является CA, тогда в представление будет выведена показанная ниже разметка:

```
<select id="Country" name="Country">
    <option value="MX">Mexico</option>
    <option selected="selected" value="CA">Canada</option>
    <option value="US">USA</option>
</select>
```

Вспомогательные функции дескрипторов для проверки достоверности

Вспомогательные функции дескрипторов для сообщения проверки достоверности и для сводки по проверке достоверности в точности отражают вспомогательные функции HTML с именами `Html.ValidationMessageFor()` и `Html.ValidationSummaryFor()`. Первая применяется к HTML-дескриптору `` для отдельного свойства модели, а вторая — к HTML-дескриптору `<div>` для целой модели. Сводка по проверке достоверности поддерживает варианты All (все ошибки), `ModelOnly` (ошибки только модели, но не свойств модели) и None (никаких ошибок).

Вспомните вспомогательные функции дескрипторов для проверки достоверности из `EditorTemplate` в файле `Car.cshtml` (выделены полужирным):

```
<div asp-validation-summary="All" class="text-danger"></div>
<div class="form-group">
    <label asp-for="PetName" class="col-form-label"></label>
    <input asp-for="PetName" class="form-control" />
    <span asp-validation-for="PetName" class="text-danger"></span>
</div>

<div class="form-group">
    <label asp-for="MakeId" class="col-form-label"></label>
    <select asp-for="MakeId" class="form-control"
        asp-items="ViewBag.MakeId"></select>
</div>

<div class="form-group">
    <label asp-for="Color" class="col-form-label"></label>
    <input asp-for="Color" class="form-control"/>
    <span asp-validation-for="Color" class="text-danger"></span>
</div>
```

Эти вспомогательные функции дескрипторов будут отображать ошибки модели, возникшие во время привязки и проверки достоверности, как показано на рис. 31.3.

Вспомогательная функция дескриптора для среды

Вспомогательная функция дескриптора для среды (`<environment>`) обычно используется для условной загрузки файлов JavaScript и CSS (или подходящей разметки) на основе среды, в которой запущен сайт. Откройте частичное представление `_Head.cshtml` и модифицируйте разметку следующим образом:

```
<meta charset="utf-8" />
<meta name="viewport"
      content="width=device-width,
      initial-scale=1.0" />
<title>@ViewData["Title"]
 - AutoLot.Mvc</title>
<environment include="Development">
  <link rel="stylesheet"
        href="~/lib/bootstrap/dist/css/bootstrap.css" />
</environment>
<environment exclude="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
</environment>
<link rel="stylesheet" href="~/css/site.css" />
```

В первой вспомогательной функции дескриптора для среды применяется атрибут `include="Development"`, чтобы включить содержащиеся файлы, когда среда установлена в Development. В таком случае загружается неминифицированная версия Bootstrap. Во второй вспомогательной функции дескриптора для среды используется атрибут `exclude="Development"`, чтобы задействовать содержащиеся файлы, когда среда отличается от Development. В таком случае загружается минифицированная версия Bootstrap. Файл `site.css` остается тем же самым в среде Development и других средах, поэтому он загружается за пределами вспомогательной функции дескриптора для среды.

Теперь модифицируйте частичное представление `_JavaScriptFiles.cshtml`, как показано ниже (обратите внимание, что файлы в разделе Development больше не имеют расширения `.min`):

```
<environment include="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
</environment>
<environment exclude="Development">
  <script src="~/lib/jquery/dist/jquery.min.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js">
    </script>
</environment>
<script src="~/js/site.js" asp-append-version="true"></script>
```

Create a New Car

- The Pet Name field is required.
- The Color field is required.

Pet Name

Validation Summary

The Pet Name field is required.

Make

BMW

Input Validation

Color

The Color field is required.

[Create +](#) | [Back to List](#)

Рис. 31.3. Вспомогательные функции дескрипторов для проверки достоверности в действии

Вспомогательная функция дескриптора для ссылки

Вспомогательная функция дескриптора для ссылки (`<link>`) имеет атрибуты, применяемые с локальными и удаленными файлами. Атрибут `asp-append-version`, используемый с локальными файлами, добавляет хеш-значение для файла как параметр строки запроса в URL, который отправляется браузеру. При изменении файла изменяется и хеш-значение, обновляя посылаемый браузеру URL. Поскольку ссылка изменилась, браузер очищает кеш от этого файла и перезагружает его. Модифицируйте дескрипторы ссылок для `bootstrap.css` и `site.css` в файле `_Head.cshtml` следующим образом:

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
    asp-appendversion="true"/>
</environment>
<environment exclude="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
</environment>
<link rel="stylesheet" href("~/css/site.css" asp-append-version="true")/>
```

Ссылка, отправляемая браузеру для файла `site.css`, теперь выглядит так (ваше хеш-значение будет другим):

```
<link href="/css/site.css?v=v9cmzjNgxPHiyLIrNom5fw3tZj3TNT2QD7a0hBrSa4U"
  rel="stylesheet">
```

При загрузке файлов CSS из сети доставки содержимого вспомогательные функции дескрипторов предоставляют механизм тестирования, позволяющий удостовериться в том, что файл был загружен надлежащим образом. Тест ищет конкретное значение для свойства в определенном классе CSS, и если свойство не дает совпадения, то вспомогательная функция дескриптора загрузит запасной файл. Модифицируйте раздел `<environment exclude="Development">` в файле `_Head.cshtml`, как показано ниже:

```
<environment exclude="Development">
  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallbacktest-value="absolute"
    crossorigin="anonymous"
    integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/
iJTQUOhcWr7x9JvoRxT2M Zw1T"/>
</environment>
```

Вспомогательная функция дескриптора для сценария

Вспомогательная функция дескриптора для сценария (`<script>`) похожа на вспомогательную функцию дескриптора для ссылки с настройками очистки кеша и перехода на запасной вариант загрузки из сети доставки содержимого. Атрибут `asp-append-version` работает для сценариев точно так же, как для ссылок на таблицы стилей. Атрибуты `asp-fallback-*` также применяются с источниками фай-

лов в сети доставки содержимого. Атрибут `asp-fallback-test` просто проверяет достоверность кода JavaScript и в случае неудачи загружает файл из запасного источника.

Обновите частичное представление `_JavaScriptFiles.cshtml`, чтобы использовать очистку кеша и переход на запасной вариант загрузки из сети доставки содержимого (обратите внимание, что шаблон MVC уже содержит атрибут `asp-append-version` в дескрипторе `<script>` для `site.js`):

```

<environment include="Development">
  <script src "~/lib/jquery/dist/jquery.js"
    asp-append-version="true"></script>
  <script src "~/lib/bootstrap/dist/js/bootstrap.bundle.js"
    asp-append-version="true"></script>
</environment>
<environment exclude="Development">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.5.1/
jquery.min.js"
    asp-fallback-src "~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha256-FgCb/KJQ1LNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
</script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/
bootstrap.bundle.min.js"
    asp-fallback-src "~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.
jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-xRywqdh3PHs8keKZN+8zzc5TX0GRTLCCmivcbNJWm2rs5C
8PRhcEn3czEjhAO9o">
</script>
</environment>
<script src("~/js/site.js" asp-append-version="true")></script>

```

Частичное представление `_ValidationScriptsPartial.cshtml` необходимо обновить с применением вспомогательных функций дескрипторов для среды и сценариев:

```

<environment include="Development">
  <script src "~/lib/jquery-validation/dist/jquery.validate.js"
    asp-append-version="true"></script>
  <script src "~/lib/jquery-validation-unobtrusive/jquery.validate.
unobtrusive.js"
    asp-append-version="true"></script>
</environment>
<environment exclude="Development">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.19.1/jquery.validate.min.js"
    asp-fallback-src "~/lib/jquery-validation/dist/jquery.validate.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator"
    crossorigin="anonymous"
    integrity="sha256-F6h55Qw6sweK+t7SiOJX+2bpSAa3b/fnlrVCJvmEj1A=">
</script>

```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"
    asp-fallback-src="~/lib/jquery-validation-unobtrusive/
jquery.validate.unobtrusive.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator &&
window.jQuery.validator.
    unobtrusive"
    crossorigin="anonymous"
    integrity="sha256-9GycpJnliUjJDVDqP0UEu/bsm9U+3dnQUH8+3W10vkY=">
</script>
</environment>
```

Вспомогательная функция дескриптора для изображения

Вспомогательная функция дескриптора для изображения (``) предоставляет атрибут `asp-append-version`, который работает точно так же, как во вспомогательных функциях дескрипторов для ссылки и сценария.

Специальные вспомогательные функции дескрипторов

Специальные вспомогательные функции дескрипторов могут помочь избавиться от повторяющегося кода. В проекте `AutoLot.Mvc` специальные вспомогательные функции дескрипторов заменят HTML-разметку, используемую для навигации между экранами CRUD для `Car`.

Подготовительные шаги

Специальные вспомогательные функции дескрипторов задействуют `UrlHelperFactory` и `IActionContextAccessor` для ссылок на основе маршрутизации. Кроме того, будет добавлен расширяющий метод для типа `string`, чтобы удалять суффикс `Controller` из имен контроллеров.

Обновление `Startup.cs`

Для создания экземпляра `UrlFactory` класса, производного не от класса `Controller`, в коллекцию служб потребуется добавить `IActionContextAccessor`. Начните с добавления в файл `Startup.cs` следующих пространств имен:

```
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.Extensions.DependencyInjection.Extensions;
```

Затем добавьте в метод `ConfigureServices()` такую строку:

```
services.TryAddSingleton<IActionContextAccessor, ActionContextAccessor>();
```

Создание расширяющего метода для типа `string`

При обращении к именам контроллеров в коде инфраструктуре ASP.NET Core довольно часто требуется низкоуровневое строковое значение, не содержащее суффикса `Controller`, что препятствует использованию метода `nameof()` без последующего вызова `string.Replace()`. Со временем задача становится утомительной, поэтому для ее решения будет создан расширяющий метод для типа `string`.

Создайте в проекте AutoLot.Services новый каталог по имени Utilities и добавьте в него файл StringExtensions.cs со статическим классом StringExtensions. Модифицируйте код, добавив расширяющий метод RemoveController():

```
using System;
namespace AutoLot.Mvc.Extensions
{
    public static class StringExtensions
    {
        public static string RemoveController(this string original)
            => original.Replace("Controller", "", StringComparison.OrdinalIgnoreCase);
    }
}
```

Создание базового класса

Создайте в проекте AutoLot.Mvc новый каталог по имени TagHelpers и внутри него каталог Base. Добавьте в каталог Base файл класса ItemLinkTagHelperBase.cs, сделайте класс ItemLinkTagHelperBase открытым и абстрактным, а также унаследованным от класса TagHelper. Приведите код класса к следующему виду:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Services.Utilities;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AutoLot.Mvc.TagHelpers.Base
{
    public abstract class ItemLinkTagHelperBase : TagHelper
    {
    }
}
```

Добавьте конструктор, который принимает экземпляры реализаций IActionContextAccessor и IUrlHelperFactory. Используйте UrlHelperFactory с ActionContextAccessor, чтобы создать экземпляр реализации IUrlHelper, и сохраните его в переменной уровня класса. Вот необходимый код:

```
protected readonly IUrlHelper UrlHelper;
protected ItemLinkTagHelperBase(IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
{
    UrlHelper = urlHelperFactory.GetUrlHelper(contextAccessor.ActionContext);
}
```

Добавьте открытое свойство для хранения Id элемента:

```
public int? ItemId { get; set; }
```

При вызове вспомогательной функции дескриптора вызывается метод Process(), принимающий два параметра, TagHelperContext и TagHelperOutput. Параметр TagHelperContext применяется для получения остальных атрибутов дескриптора и словаря объектов, которые используются с целью взаимодействия с другими вспомо-

гательными функциями дескрипторов, нацеленными на дочерние элементы. Параметр TagHelperOutput применяется для создания визуализированного вывода. Поскольку это базовый класс, создайте метод по имени BuildContent(), который производные классы смогут вызывать из метода Process(). Добавьте следующий код:

```
protected void BuildContent(TagHelperOutput output, string actionName,
    string className, string displayText, string fontAwesomeName)
{
    output.TagName = "a"; // Заменить <item-list> дескриптором <a>.
    var target = (ItemId.HasValue)
        ? UrlHelper.Action(actionName,
            nameof(CarsController).RemoveController(), new { id = ItemId })
        : UrlHelper.Action(actionName,
            nameof(CarsController).RemoveController());
    output.Attributes.SetAttribute("href", target);
    output.Attributes.Add("class", className);
    output.Content.AppendHtml($"<i class=""fas fa-{fontAwesomeName}""></i>");
}
```

В предыдущем код присутствует ссылка на набор инструментов для значков и шрифтов Font Awesome, который будет добавлен в проект позже в главе.

Вспомогательная функция дескриптора для вывода сведений об элементе

Создайте в каталоге TagHelpers новый файл класса по имени ItemDetailsTagHelper.cs. Сделайте класс ItemDetailsTagHelper открытым и унаследованным от класса ItemLinkTagHelperBase. Добавьте в новый файл показанный ниже код:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Mvc.TagHelpers.Base;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AutoLot.Mvc.TagHelpers
{
    public class ItemDetailsTagHelper : ItemLinkTagHelperBase
    {
    }
}
```

Добавьте открытый конструктор, который принимает обязательные экземпляры и передает их конструктору базового класса:

```
public ItemDetailsTagHelper(
    IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
    : base(contextAccessor, urlHelperFactory) { }
```

Переопределите метод Process(), чтобы вызывать метод BuildContent() базового класса:

```
public override void Process(TagHelperContext context,
    TagHelperOutput output)
```

```
{
    BuildContent(output, nameof(CarsController.Details),
                "text-info", "Details", "info-circle");
}
```

Код создает ссылку Details (Детали) с изображением значка информации из Font Awesome. Чтобы не возникали ошибки при компиляции, добавьте в CarsController базовый метод Details():

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

Вспомогательная функция дескриптора для удаления элемента

Создайте в каталоге TagHelpers новый файл класса по имени ItemDeleteTagHelper.cs. Сделайте класс ItemDeleteTagHelper открытым и унаследованным от класса ItemLinkTagHelperBase. Добавьте в новый файл следующий код:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Mvc.TagHelpers.Base;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AutoLot.Mvc.TagHelpers
{
    public class ItemDeleteTagHelper : ItemLinkTagHelperBase
    {
    }
}
```

Добавьте открытый конструктор, который принимает обязательные экземпляры и передает их конструктору базового класса:

```
public ItemDeleteTagHelper(
    IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
    : base(contextAccessor, urlHelperFactory) { }
```

Переопределите метод Process(), чтобы вызывать метод BuildContent() базового класса:

```
public override void Process(TagHelperContext context,
                            TagHelperOutput output)
{
    BuildContent(output, nameof(CarsController.Delete),
                "text-danger", "Delete", "trash");
}
```

Код создает ссылку Delete (Удалить) с изображением значка мусорного ящика из Font Awesome. Чтобы не возникали ошибки при компиляции, добавьте в CarsController базовый метод Delete():

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

Вспомогательная функция дескриптора для редактирования сведений об элементе

Создайте в каталоге TagHelpers новый файл класса по имени ItemEditTagHelper.cs. Сделайте класс ItemEditTagHelper открытым и унаследованным от класса ItemLinkTagHelperBase. Добавьте в новый файл показанный ниже код:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Mvc.TagHelpers.Base;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AutoLot.Mvc.TagHelpers
{
    public class ItemEditTagHelper : ItemLinkTagHelperBase
    {
    }
}
```

Добавьте открытый конструктор, который принимает обязательные экземпляры и передает их конструктору базового класса:

```
public ItemEditTagHelper(
    IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
    : base(contextAccessor, urlHelperFactory) { }
```

Переопределите метод Process(), чтобы вызывать метод BuildContent() базового класса:

```
public override void Process(TagHelperContext context,
                             TagHelperOutput output)
{
    BuildContent(output, nameof(CarsController.Edit),
                "text-warning", "Edit", "edit");
}
```

Код создает ссылку Edit (Редактировать) с изображением значка карандаша из Font Awesome. Чтобы не возникали ошибки при компиляции, добавьте в CarsController базовый метод Edit():

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

Вспомогательная функция дескриптора для создания элемента

Создайте в каталоге TagHelpers новый файл класса по имени ItemCreateTagHelper.cs. Сделайте класс ItemCreateTagHelper открытым и унаследованным от класса ItemLinkTagHelperBase. Добавьте в новый файл следующий код:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Mvc.TagHelpers.Base;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
```

```
using Microsoft.AspNetCore.Razor.TagHelpers;
namespace AutoLot.Mvc.TagHelpers
{
    public class ItemCreateTagHelper : ItemLinkTagHelperBase
    {
    }
}
```

Добавьте открытый конструктор, который принимает обязательные экземпляры и передает их конструктору базового класса:

```
public ItemCreateTagHelper(
    IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
: base(contextAccessor, urlHelperFactory) { }
```

Переопределите метод `Process()`, чтобы вызывать метод `BuildContent()` базового класса:

```
public override void Process(TagHelperContext context,
                            TagHelperOutput output)
{
    BuildContent(output, nameof(CarsController.Create),
                "text-success", "Create new", "plus");
}
```

Код создает ссылку `Create new` (Создать) с изображением значка плюса из Font Awesome.

Вспомогательная функция дескриптора для вывода списка элементов

Создайте в каталоге `TagHelpers` новый файл класса по имени `ItemListTagHelper.cs`. Сделайте класс `ItemListTagHelper` открытым и унаследованным от класса `ItemLinkTagHelperBase`. Добавьте в новый файл показанный ниже код:

```
using AutoLot.Mvc.Controllers;
using AutoLot.Mvc.TagHelpers.Base;
using Microsoft.AspNetCore.Mvc.Infrastructure;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AutoLot.Mvc.TagHelpers
{
    public class ItemListTagHelper : ItemLinkTagHelperBase
    {
    }
}
```

Добавьте открытый конструктор, который принимает обязательные экземпляры и передает их конструктору базового класса:

```
public ItemListTagHelper(
    IActionContextAccessor contextAccessor,
    IUrlHelperFactory urlHelperFactory)
: base(contextAccessor, urlHelperFactory) { }
```

Переопределите метод `Process()`, чтобы вызывать метод `BuildContent()` базового класса:

```
public override void Process(TagHelperContext context,
                             TagHelperOutput output)
{
    BuildContent(output, nameof(CarsController.Index),
                 "text-default", "Back to List", "list");
}
```

Код создает ссылку `Back to List` (Список) с изображением значка списка из `Font Awesome`. Чтобы не возникали ошибки при компиляции, добавьте в `CarsController` базовый метод `Index()`:

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

Обеспечение видимости специальных вспомогательных функций дескрипторов

Чтобы сделать специальные вспомогательные функции дескрипторов видимыми, потребуется выполнить команду `@addTagHelper` для представлений, которые используют эти вспомогательные функции дескрипторов, или поместить ее в файл `_ViewImports.cshtml`. Откройте файл `_ViewImports.cshtml` в каталоге `Views` и добавьте в него следующую строку:

```
@addTagHelper *, AutoLot.Mvc
```

Вспомогательные функции HTML

Вспомогательные функции HTML из ASP.NET MVC по-прежнему поддерживаются, а некоторые из них применяются довольно широко и перечислены в табл. 31.5.

Таблица 31.5. Часто используемые вспомогательные функции HTML

Вспомогательная функция HTML	Описание
<code>Html.DisplayFor()</code>	Отображает объект, определяемый выражением
<code>Html.DisplayForModel()</code>	Отображает модель с применением стандартного или специального шаблона
<code>Html.DisplayNameFor()</code>	Получает отображаемое имя, если оно существует, либо имя свойства, если отображаемое имя отсутствует
<code>Html.EditorFor()</code>	Отображает редактор для объекта, определяемого выражением
<code>Html.EditorForModel()</code>	Отображает редактор для модели с использованием стандартного или специального шаблона

Вспомогательная функция `DisplayFor()`

Вспомогательная функция `DisplayFor()` отображает объект, определяемый выражением. Если для отображаемого типа существует шаблон отображения, тогда он будет применяться при создании HTML-разметки, представляющей элемент. Например, если моделью представления является сущность `Car`, то информацию о производителе автомобиля можно отобразить следующим образом:

```
@Html.DisplayFor(x=>x.MakeNavigation);
```

Если в каталоге `DisplayTemplates` присутствует представление по имени `Make.cshtml`, тогда оно будет использоваться для визуализации значений (вспомните, что поиск имени шаблона базируется на типе объекта, а не на имени его свойства). Если представление по имени `ShowMake.cshtml` (например) существует, то оно будет применяться для визуализации объекта с помощью приведенного ниже вызова:

```
@Html.DisplayFor(x=>x.MakeNavigation, "ShowMake");
```

В случае если шаблон не указан и отсутствует представление с именем класса, тогда для создания HTML-разметки, подлежащей отображению, используется рефлексия.

Вспомогательная функция `DisplayForModel()`

Вспомогательная функция `DisplayForModel()` отображает модель для представления. Если для отображаемого типа существует шаблон отображения, то он будет применяться при создании HTML-разметки, представляющей элемент. Продолжая предыдущий пример представления с сущностью `Car` в качестве модели, полную информацию `Car` можно отобразить следующим образом:

```
@Html.DisplayForModel();
```

Как и в случае со вспомогательной функцией `DisplayFor()`, если существует шаблон отображения, имеющий имя типа, тогда он будет использоваться. Можно также применять именованные шаблоны. Скажем, для отображения сущности `Car` с помощью шаблона отображения `CarWithColors.html` необходимо использовать такой вызов:

```
@Html.DisplayForModel("CarWithColors");
```

Если шаблон не указан и отсутствует представление с именем класса, то для создания HTML-разметки, подлежащей отображению, используется рефлексия.

Вспомогательные функции `EditorFor()` и `EditorForModel()`

Вспомогательные функции `EditorFor()` и `EditorForModel()` работают аналогично соответствующим вспомогательным функциям для отображения, но с тем отличием, что шаблоны ищутся в каталоге `EditorTemplates` и вместо представления объекта, предназначенного только для чтения, отображаются HTML-формы редакторов.

Управление библиотеками клиентской стороны

До завершения представлений нужно обновить библиотеки клиентской стороны (CSS и JavaScript). Проект диспетчера библиотек `LibraryManager` (первоначаль-

но разрабатываемый Мэдсом Кристенсеном) теперь является частью Visual Studio (VS2019) и также доступен в виде глобального инструмента .NET Core. Для извлечения инструментов CSS и JavaScript из CDNJS.com, UNPKG.com, jsDelivr.com или файловой системы в LibraryManager используется простой файл JSON.

Установка диспетчера библиотек как глобального инструмента .NET Core

Диспетчер библиотек встроен в Visual Studio. Чтобы установить его как глобальный инструмент .NET Core, введите следующую команду:

```
dotnet tool install --global Microsoft.Web.LibraryManager.Cli  
--version 2.1.113
```

Текущая версия диспетчера библиотек доступна по ссылке <https://www.nuget.org/packages/Microsoft.Web.LibraryManager.Cli/>.

Добавление в проект AutoLot.Mvc библиотек клиентской стороны

При создании проекта AutoLot.Mvc (с помощью Visual Studio или командной строки .NET Core CLI) в каталог wwwroot\lib было установлено несколько файлов JavaScript и CSS. Удалите каталог lib вместе со всеми содержащимися в нем файлами, т.к. все они будут заменены диспетчером библиотек.

Добавление файла libman.json

Файл libman.json управляет тем, что именно устанавливается, из каких источников и куда попадают установленные файлы.

Visual Studio

Если вы работаете в Visual Studio, тогда щелкните правой кнопкой мыши на имени проекта AutoLot.Mvc и выберите в контекстном меню пункт Manage Client-Side Libraries (Управлять библиотеками клиентской стороны), в результате чего в корневой каталог проекта добавится файл libman.json. В Visual Studio также есть возможность связать диспетчер библиотек с процессом MSBuild. Щелкните правой кнопкой мыши на имени файла libman.json и выберите в контекстном меню пункт Enable restore on build (Включить восстановление при сборке). Вам будет предложено разрешить другому пакету NuGet (Microsoft.Web.LibraryManager.Build) восстановиться в проекте. Разрешите установку пакета.

Командная строка

Создайте новый файл libman.json посредством следующей команды (она устанавливает CDNJS.com в качестве стандартного поставщика):

```
libman init --default-provider cdnjs
```

Обновление файла libman.json

Для поиска библиотек, подлежащих установке, сеть доставки содержимого CDNJS.com предлагает удобный для человека API-интерфейс. Список всех доступных библиотек можно просмотреть по следующему URL:

<https://api.cdnjs.com/libraries?output=human>

Найдя библиотеку, которую вы хотите установить, модифицируйте URL, указав имя библиотеки из списка, чтобы увидеть ее версии и файлы для каждой версии. Например, для просмотра всех доступных версий и файлов jQuery используйте такую ссылку:

```
https://api.cdnjs.com/libraries/jquery?output=human
```

После выбора версии и файлов для установки добавьте имя библиотеки (плюс версию), место назначения (обычно `wwwroot/lib/<ИмяБиблиотеки>`) и файлы, которые требуется загрузить. Скажем, чтобы загрузить jQuery, введите в массив JSON библиотеки следующий код:

```
{
  "library": "jquery@3.5.1",
  "destination": "wwwroot/lib/jquery",
  "files": [ "jquery.js" ]
},
```

Ниже приведено полное содержимое файла `libman.json`, где указаны все файлы, необходимые для разрабатываемого приложения:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "defaultDestination": "wwwroot/lib",
  "libraries": [
    {
      "library": "jquery@3.5.1",
      "destination": "wwwroot/lib/jquery",
      "files": [ "jquery.js", "jquery.min.js" ]
    },
    {
      "library": "jquery-validate@1.19.2",
      "destination": "wwwroot/lib/jquery-validation",
      "files": [ "jquery.validate.js", "jquery.validate.min.js",
        "additional-methods.js",
        "additional-methods.min.js" ]
    },
    {
      "library": "jquery-validation-unobtrusive@3.2.11",
      "destination": "wwwroot/lib/jquery-validation-unobtrusive",
      "files": [ "jquery.validate.unobtrusive.js",
        "jquery.validate.unobtrusive.min.js" ]
    },
    {
      "library": "twitter-bootstrap@4.5.3",
      "destination": "wwwroot/lib/bootstrap",
      "files": [
        "css/bootstrap.css",
        "js/bootstrap.bundle.js",
        "js/bootstrap.js"
      ]
    },
    {
      "library": "font-awesome@5.15.1",
      "destination": "wwwroot/lib/font-awesome/",
```

```

    "files": [
      "js/all.js",
      "css/all.css",
      "sprites/brands.svg",
      "sprites/regular.svg",
      "sprites/solid.svg",
      "webfonts/fa-brands-400.eot",
      "webfonts/fa-brands-400.svg",
      "webfonts/fa-brands-400.ttf",
      "webfonts/fa-brands-400.woff",
      "webfonts/fa-brands-400.woff2",
      "webfonts/fa-regular-400.eot",
      "webfonts/fa-regular-400.svg",
      "webfonts/fa-regular-400.ttf",
      "webfonts/fa-regular-400.woff",
      "webfonts/fa-regular-400.woff2",
      "webfonts/fa-solid-900.eot",
      "webfonts/fa-solid-900.svg",
      "webfonts/fa-solid-900.ttf",
      "webfonts/fa-solid-900.woff",
      "webfonts/fa-solid-900.woff2"
    ]
  }
]
}

```

На заметку! Вскоре будет объяснена причина отсутствия в списке минифицированных файлов.

После сохранения libman.json (в Visual Studio) файлы будут загружены в каталог wwwroot\lib проекта. Если же вы работаете в командной строке, тогда введите следующую команду, чтобы перезагрузить все файлы:

```
libman restore
```

Доступны дополнительные параметры командной строки, которые можно просмотреть с помощью команды libman -h.

Обновление ссылок на файлы JavaScript и CSS

С переходом на диспетчер библиотек местоположение многих файлов JavaScript и CSS изменилось. Файлы Bootstrap и jQuery были загружены в каталог \dist. Кроме того, в приложение был добавлен набор инструментов для значков и шрифтов Font Awesome.

Местоположение файлов Bootstrap необходимо изменить на ~/lib/bootstrap/css вместо ~/lib/bootstrap/dist/css. Добавьте Font Awesome в конец, прямо перед site.css. Модифицируйте файл _Head.cshtml, как показано ниже:

```

<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>@ ViewData["Title"] - AutoLot.Mvc</title>
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css"
        asp-appendversion="true"/>
</environment>

```

```

<environment exclude="Development">
  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/css/bootstrap.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position" asp-fallback-test-
    value="absolute"
    crossorigin="anonymous"
    integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/
iJTQUOhcWr7x9JvoRxT2MZw1T"/>
</environment>
<link rel="stylesheet" href="~/lib/font-awesome/css/all.css"
  asp-append-version="true"/>
<link rel="stylesheet" href("~/css/site.css" asp-append-version="true")/>

```

Далее модифицируйте файл _JavaScriptFiles.cshtml, удалив \dist из местоположений jQuery и Bootstrap:

```

<environment include="Development">
  <script src="~/lib/jquery/jquery.js" asp-append-version="true">
  </script>
  <script src="~/lib/bootstrap/js/bootstrap.bundle.js"
    asp-append-version="true"></script>
</environment>
<environment exclude="Development">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.5.1/
jquery.min.js"
    asp-fallback-src="~/lib/jquery/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha256-FgCb/KJQ1LNfOu91ta32o/NMZXltwRo8QtmkMRdAu8=">
  </script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/
bootstrap.bundle.min.js"
    asp-fallback-src="~/lib/bootstrap/js/bootstrap.bundle.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-xrRywqdh3PHs8keKZN+8zzc5TX0GRTLCCmivcbNJWm2rs5C
8PRhcEn3czEjhAO9o">
  </script>
</environment>
<script src "~/js/site.js" asp-append-version="true"></script>

```

Финальное изменение связано с обновлением местоположений jquery.validate в частичном представлении _ValidationScriptsPartial.cshtml:

```

<environment include="Development">
  <script src="~/lib/jquery-validation/jquery.validate.js"
    asp-append-version="true"></script>
  <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.
unobtrusive.js"
    asp-append-version="true"></script>
</environment>

```

```

<environment exclude="Development">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.19.1/jquery.validate.min.js"
    asp-fallback-src="~/lib/jquery-validation/jquery.validate.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator"
    crossorigin="anonymous"
    integrity="sha256-F6h55Qw6sweK+t7SiOJX+2bpSAa3b/fnlrVCJvmEj1A="
  </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validation-unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"
    asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.
validate.unobtrusive.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator &&
window.jQuery.validator.unobtrusive" crossorigin="anonymous"
    integrity="sha256-9GycpJnliUjJDVDqP0UEu/bsm9U+3dnQUH8+3W10vky=">
  </script>
</environment>

```

Завершение работы над представлениями CarsController и Cars

В этом разделе будет завершена работа над представлениями CarsController и Cars. Если вы установите в true флаг RebuildDatabase внутри файла appsettings.development.json, тогда любые изменения, внесенные вами во время тестирования этих представлений, будут сбрасываться при следующем запуске приложения.

Класс CarsController

Класс CarsController является центральной точкой приложения AutoLot.Mvc, обладая возможностями создания, чтения, обновления и удаления. В этой версии CarsController напрямую используется уровень доступа к данным. Позже в главе вы создадите еще одну версию CarsController, в которой для доступа к данным будет применяться служба AutoLot.Api.

Приведите операторы using в классе CarsController к следующему виду:

```

using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Models.Entities;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;

```

Ранее вы добавили класс контроллера с маршрутом. Теперь наступило время добавить экземпляры реализаций ICarRepo и IAppLogging<CarsController> через внедрение зависимостей. Добавьте две переменные уровня класса для хранения этих экземпляров, а также конструктор, который будет внедрять оба экземпляра:

```

private readonly ICarRepo _repo;
private readonly IAppLogging<CarsController> _logging;
public CarsController(ICarRepo repo, IAppLogging<CarsController> logging)
{
  _repo = repo;
  _logging = logging;
}

```

Частичное представление списка автомобилей

Списковые представления (одно для целого реестра автомобилей и одно для списка автомобилей по производителям) совместно используют частичное представление. Создайте в каталоге Views\Cars новый каталог по имени Partials и добавьте в него файл представления _CarListPartial.cshtml, очистив его содержимое. Установите I Enumerable<Car> в качестве типа (его не нужно указывать полностью, поскольку в файл _ViewImports.cshtml добавлено пространство имен AutoLot.Models.Entities):

```
@model IEnumerable<Car>
```

Далее добавьте блок кода Razor с набором булевых переменных, которые указывают, должны ли отображаться производители. Когда частичное представление _CarListPartial.cshtml применяется полным реестром автомобилей, производители будут показаны, а когда отображаются автомобили только одного производителя, то поле Make должно быть скрыто:

```
@{
    var showMake = true;
    if (bool.TryParse(ViewBag.ByMake?.ToString(), out bool byMake))
    {
        showMake = !byMake;
    }
}
```

В следующей разметке ItemCreateTagHelper используется для создания ссылки на метод Create() типа HttpGet. В случае применения специальных вспомогательных функций дескрипторов имя указывается с использованием “шашлычного” стиля в нижнем регистре, т.е. суффикс TagHelper отбрасывается, а каждое слово в стиле Pascal приводится к нижнему регистру и отделяется символом переноса (что похоже на шашлык):

```
<p>
    <item-create></item-create>
</p>
```

Для настройки таблицы и ее заголовков применяется вспомогательная функция HTML, посредством которой получаются значения DisplayName, связанные с каждым свойством. Для DisplayName будет выбираться значение атрибута Display или DisplayName, и если он не установлен, то будет использоваться имя свойства. В следующем разделе применяется блок кода Razor для отображения информации о производителе на основе ранее установленной переменной уровня представления:

```
<table class="table">
    <thead>
        <tr>
            @if (showMake)
            {
                <th>
                    @Html.DisplayNameFor(model => model.MakeId)
                </th>
            }
            <th>
                @Html.DisplayNameFor(model => model.Color)
            </th>
        
```

```

<th>
    @Html.DisplayNameFor(model => model.PetName)
</th>
<th></th>
</tr>
</thead>

```

В последнем разделе производится проход по записям и их отображение с использованием вспомогательной функции HTML по имени `DisplayFor()`. Эта вспомогательная функция HTML ищет шаблон отображения с именем, соответствующим типу свойства, и если шаблон не обнаруживается, то разметка создается стандартным образом. Для каждого свойства объекта также выполняется поиск шаблона отображения, который применяется при его наличии. Например, если `Car` имеет свойство `DateTime`, то для него будет использоваться показанный ранее в главе шаблон `DisplayTemplate`.

С следующем блоке также задействованы специальные вспомогательные функции дескрипторов `item-edit`, `item-details` и `item-delete`, которые были добавлены ранее. Обратите внимание, что при передаче значений открытому свойству специальней вспомогательной функции имя свойства указывается с применением "шашлычного" стиля в нижнем регистре и добавляется к дескриптору в виде атрибута:

```

<tbody>
    @foreach (var item in Model)
    {
        <tr>
            @if (showMake)
            {
                <td>
                    @Html.DisplayFor(modelItem => item.MakeNavigation.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Color)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.PetName)
                </td>
                <td>
                    <item-edit item-id="@item.Id"></item-edit> |
                    <item-details item-id="@item.Id"></item-details> |
                    <item-delete item-id="@item.Id"></item-delete>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Представление Index

При наличии частичного представления `_CarListPartial` представление `Index` будет небольшим. Создайте в каталоге `Views\Cars` новый файл представления по имени `Index.cshtml`. Удалите весь генерированный код и добавьте следующую разметку:

```
@model IEnumerable<Car>
 @{
    ViewData["Title"] = "Index";
}
<h1>Vehicle Inventory</h1>
<partial name="Partials/_CarListPartial" model="@Model"/>
```

Частичное представление `_CarListPartial` вызывается со значением модели содержащего представления (`IEnumerable<Car>`), которое передается с помощью атрибута `model`. В итоге модель частичного представления устанавливается в объект, переданный вспомогательной функции дескриптора `<partial>`.

Чтобы взглянуть на представление `Index`, модифицируйте метод `Index()` класса `CarsController`, как показано ниже:

```
[Route("/{controller}")]
[Route("/{controller}/{action}")]
public IActionResult Index()
    => View(_repo.GetAllIgnoreQueryFilters());
```

Запустив приложение и перейдя по ссылке `https://localhost:5001/Cars/Index`, вы увидите список автомобилей (рис. 31.4).

Make	Color	Pet Name	
BMW	Black	Bimmer	Edit Details Delete
Yugo	Brown	Brownie	Edit Details Delete
Yugo	Yellow	Clunker	Edit Details Delete
BMW	Green	Hank	Edit Details Delete
VW	Rust	Lemon	Edit Details Delete
Saab	Black	Mel	Edit Details Delete
Pinto	Black	Pete	Edit Details Delete
BMW	Pink	Pinky	Edit Details Delete
Ford	Rust	Rusty	Edit Details Delete
VW	Black	Zippy	Edit Details Delete

Рис. 31.4. Страница реестра автомобилей

В правой части списка отображаются специальные вспомогательные функции дескрипторов.

Представление ByMake

Представление `ByMake` похоже на `Index`, но настраивает частичное представление так, что информация о производителе отображается только в заголовке страницы. Создайте в каталоге `Views\Cars` новый файл представления по имени `ByMake.cshtml`. Удалите весь сгенерированный код и добавьте следующую разметку:

```
@model IEnumerable<Car>
{
    ViewData["Title"] = "Index";
}
<h1>Vehicle Inventory for @ViewBag.MakeName</h1>
{
    var mode = new ViewDataDictionary(ViewData) {{"ByMake", true}};
}
<partial name="Partials/_CarListPartial" model="Model" view-data="@mode"/>
```

Отличия заметить легко. Здесь создается экземпляр ViewDataDictionary, содержащий свойство ByMake из ViewBag, который затем вместе с моделью передается частичному представлению, что позволяет скрыть информацию о производителе.

Метод действия для этого представления должен получить все автомобили с указанным значением MakeId и установить ViewBag в MakeName с целью отображения в пользовательском интерфейсе. Оба значения берутся из маршрута. Добавьте в класс CarsController новый метод действия по имени ByMake():

```
[HttpGet("/[controller]/[action]/{makeId}/{makeName}")]
public IActionResult ByMake(int makeId, string makeName)
{
    ViewBag.MakeName = makeName;
    return View(_repo.GetAllBy(makeId));
}
```

Запустив приложение и перейдя по ссылке <https://localhost:5001/Cars/1/vw>, вы увидите список, показанный на рис. 31.5.

AutoLot.Mvc Home Razor Syntax Privacy

Vehicle Inventory for VW

Create New

Color	Pet Name	
Black	Zippy	Edit Details Delete

Рис. 31.5. Страница реестра автомобилей для конкретного производителя

Представление Details

Создайте в каталоге Views\Cars новый файл представления по имени Details.cshtml. Удалите весь генерированный код и добавьте следующую разметку:

```
@model Car
{
    ViewData["Title"] = "Details";
}
<h1>Details for @Model.PetName</h1>
@Html.DisplayForModel()
```

```
<div>
  <item-edit item-id="@Model.Id"></item-edit>
  <item-delete item-id="@Model.Id"></item-delete>
  <item-list></item-list>
</div>
```

Вспомогательная функция `@Html.DisplayForModel()` использует созданный ранее шаблон отображения (`Car.cshtml`) для вывода детальной информации об автомобиле.

Прежде чем обновлять метод действия `Details()`, добавьте вспомогательный метод по имени `GetOne()`, который будет извлекать одиночную запись `Car`:

```
internal Car GetOneCar(int? id)
  => !id.HasValue ? null : _repo.Find(id.Value);
```

Модифицируйте метод действия `Details()` следующим образом:

```
[HttpGet("{id?}")]
public IActionResult Details(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest();
  }
  var car = GetOneCar(id);
  if (car == null)
  {
    return NotFound();
  }
  return View(car);
}
```

Маршрут для метода действия `Details()` содержит необязательный параметр маршрута `id` с идентификатором автомобиля, значение которого присваивается параметру `id` метода. Обратите внимание, что у параметра маршрута есть вопросительный знак с маркером. Это указывает на необязательность параметра, почти как вопросительный знак в типе `int?` делает переменную `int` допускающей значение `null`. Если параметр не был предоставлен или оболочка службы не может отыскать автомобиль с идентификатором, заданным в параметре маршрута, тогда метод возвращает ошибку `NotFound`. В противном случае метод отправляет найденную запись `Car` представлению `Details`. Запустив приложение и перейдя по ссылке `https://localhost:5001/Cars/Details/1`, вы увидите экран, показанный на рис. 31.6.

AutoLot.Mvc Home Razor Syntax Privacy

Details for Zippy

Make	VW
Color	Black
Pet Name	Zippy

[Edit](#) [Delete](#) [Back to List](#)

Рис. 31.6. Детальная информация о конкретном автомобиле

Представление Create

Представление Create было начато ранее. Вот его полная разметка:

```

@model Car

{
    ViewData["Title"] = "Create";
}

<h1>Create a New Car</h1>
<hr/>
<div class="row">
    <div class="col-md-4">
        <form asp-controller="Cars" asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            @Html.EditorForModel()
            <div class="form-group">
                <button type="submit"
                    class="btn btn-success">Create <i class="fas faplus"></i>
                </button>&nbsp;&nbsp;|&nbsp;&nbsp;
                <item-list></item-list>
            </div>
        </form>
    </div>
</div>
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}

```

Вспомогательная функция `@Html.EditorForModel()` использует созданный ранее шаблон отображения (`Car.cshtml`) для отображения редактора сведений об автомобиле.

В разделе `Scripts` представления указано частичное представление `_ValidationScriptsPartial`. Вспомните, что в компоновке этот раздел встречается *после* загрузки jQuery. Шаблон разделов помогает гарантировать загрузку надлежащих зависимостей до загрузки самого содержимого.

Методы действий Create()

В рамках процесса создания применяются два метода действий: первый (`HttpGet`) возвращает пустое представление для ввода новой записи, а второй (`HttpPost`) отправляет значения новой записи.

Вспомогательный метод GetMakes()

Вспомогательный метод `GetMakes()` возвращает список записей Make в виде экземпляра `SelectList` и принимает в качестве параметра экземпляр реализации `IMakeRepo`:

```

internal SelectList GetMakes(IMakeRepo makeRepo)
    => new SelectList(makeRepo.GetAll(), nameof(Make.Id),
        nameof(Make.Name));

```

Метод действия Create() для GET

Метод действия `Create()` для GET помещает в словарь `ViewData` список `SelectList` с записями Make и отправляет его представлению `Create`:

```
[HttpGet]
public IActionResult
Create([FromServices] IMakeRepo makeRepo)
{
    ViewData["MakeId"] = GetMakes(makeRepo);
    return View();
}
```

Форму создания можно просмотреть по ссылке /Cars/Create (рис. 31.7).

Метод действия Create() для POST

Метод действия Create() для POST применяет неявную привязку модели для создания сущности Car из значений формы. Вот его код:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult
Create([FromServices] IMakeRepo makeRepo, Car car)
{
    if (ModelState.IsValid)
    {
        _repo.Add(car);
        return RedirectToAction(nameof(Details), new { id = car.Id });
    }
    ViewData["MakeId"] = GetMakes(makeRepo);
    return View(car);
}
```

Атрибут `HttpPost` помечает метод как конечную точку приложения для маршрута `Cars/Create`, когда запросом является POST. Атрибут `ValidateAntiForgeryToken` использует значение скрытого элемента ввода для `__RequestVerificationToken`, чтобы сократить количество атак на сайт.

Экземпляр реализации `IMakeRepo` внедряется в метод из контейнера DI. Поскольку внедрение осуществляется в метод, применяется атрибут `FromServices`. Как вы наверняка помните, атрибут `FromServices` сообщает механизму привязки о том, чтобы он не пытался привязывать этот тип, и позволяет контейнеру DI узнать о необходимости создания экземпляра класса.

Сущность `Car` неявно привязывается к данным входящего запроса. Если состояние модели (`ModelState`) допустимо, тогда сущность `Car` добавляется в базу данных и пользователь перенаправляется на метод действия `Details()` с использованием вновь созданного идентификатора `Car` в качестве параметра маршрута. Такой шаблон называется “отправка-перенаправление-получение” (Post-Redirect-Get). Пользователь выполняет отправку с помощью метода `HttpPost (Create())` и затем перенаправляется на метод `HttpGet (Details())`, что предотвращает повторную отправку браузером запроса POST, если пользователь решит обновить страницу.

Если состояние модели не является допустимым, то список `SelectList` с записями `Make` добавляется в объект `ViewData` и сущность, которая была отправлена, посыпается обратно представлению `Create`. Состояние модели тоже неявно отправляется представлению, так что могут быть отображены любые ошибки.

Create a New Car

Рис. 31.7. Представление Create

Представление Edit

Создайте в каталоге Views\Cars новый файл представления по имени Edit.cshtml. Удалите весь сгенерированный код и добавьте следующую разметку:

```
@model Car
{
    ViewData["Title"] = "Edit";
}
<h1>Edit @Model.PetName</h1>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-area="" asp-controller="Cars" asp-action="Edit"
              asp-route-id="@Model.Id">
            @Html.EditorForModel()
            <input type="hidden" asp-for="Id" />
            <input type="hidden" asp-for="TimeStamp" />
            <div class="form-group">
                <button type="submit" class="btn btn-primary">
                    Save <i class="fas fa-save"></i>
                </button>&nbsp;&nbsp;|&nbsp;
                <item-list></item-list>
            </div>
        </form>
    </div>
</div>
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

В представлении также применяется вспомогательная функция `@Html.EditorForModel()` и частичное представление `_ValidationScriptsPartial`. Однако оно еще содержит два скрытых элемента ввода для `Id` и `TimeStamp`. Они будут отправляться вместе с остальными данными формы, но не должны редактироваться пользователями. Без значений `Id` и `TimeStamp` не удалось бы сохранять изменения.

Методы действий Edit()

В рамках процесса редактирования используются два метода действий: первый (`HttpGet`) возвращает сущность, подлежащую редактированию, а второй (`HttpPut`) отправляет значения обновленной записи.

Метод действия Edit() для GET

Метод действия `Edit()` для GET получает одиночную запись `Car` с идентификатором `Id` через оболочку службы и отправляет ее представлению `Edit`:

```
[HttpGet("{id?}")]
public IActionResult Edit([FromServices] IMakeRepo makeRepo, int? id)
{
    var car = GetOneCar(id);
    if (car == null)
    {
        return NoContent();
    }
```

```
ViewData["MakeId"] = GetMakes(makeRepo);
return View(car);
}
```

Маршрут имеет необязательный параметр id, значение которого передается методу с применением параметра id. Экземпляр реализации IMakeRepo внедряется в метод и используется для создания списка SelectList записей Make. Посредством вспомогательного метода GetOneCar() получается запись Car. Если запись Car найти не удалось, тогда метод возвращает ошибку NoContent. В противном случае он добавляет список SelectList записей Make в словарь ViewData и визуализирует представление Edit.

Форму редактирования можно просмотреть по ссылке /Cars/Edit/1 (рис. 31.8).

Метод действия Edit() для POST

Метод действия Edit() для POST аналогичен методу действия Create() для POST с отличиями, описанными после кода метода:

```
[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public IActionResult Edit([FromServices] IMakeRepo makeRepo,
                           int id, Car car)
{
    if (id != car.Id)
    {
        return BadRequest();
    }
    if (ModelState.IsValid)
    {
        _repo.Update(car);
        return RedirectToAction(nameof(Details), new { id = car.Id });
    }
    ViewData["MakeId"] = GetMakes(makeRepo);
    return View(car);
}
```

Метод действия Edit() для POST принимает один обязательный параметр маршрута id. Если его значение не совпадает со значением Id реконструированной сущности Car, тогда клиенту отправляется ошибка BadRequest. Если состояние модели допустимо, то сущность обновляется, после чего пользователь перенаправляется на метод действия Details() с применением свойства Id сущности Car в качестве параметра маршрута. Здесь также используется шаблон “отправка-перенаправление-получение”.

Если состояние модели не является допустимым, то список SelectList с записями Make добавляется в объект ViewData и сущность, которая была отправлена, посыпается обратно представлению Edit. Состояние модели тоже неявно отправляется представлению, так что могут быть отображены любые ошибки.

Edit Zippy

Pet Name

Zippy

Make

VW

Color

Black

[Save](#) | [Back to List](#)

Рис. 31.8. Представление Edit

Представление Delete

Создайте в каталоге Views\Cars новый файл представления по имени Delete.cshtml. Удалите весь генерированный код и добавьте следующую разметку:

```
@model Car
@{
    ViewData["Title"] = "Delete";
}
<h1>Delete @Model.PetName</h1>
<h3>Are you sure you want to delete this car?</h3>
<div>
    @Html.DisplayForModel()
    <form asp-action="Delete">
        <input type="hidden" asp-for="Id" />
        <input type="hidden" asp-for="TimeStamp" />
        <button type="submit" class="btn btn-danger">
            Delete <i class="fas fa-trash"></i>
        </button>&nbsp;&nbsp;!&nbsp;&nbsp;
        <item-list></item-list>
    </form>
</div>
```

В представлении Delete тоже применяется вспомогательная функция @Html.DisplayForModel() и два скрытых элемента ввода для Id и TimeStamp. Это единственные поля, которые отправляются в виде данных формы.

Методы действий Delete()

В рамках процесса удаления используются два метода действий: первый (HttpGet) возвращает сущность, подлежащую удалению, а второй (HttpPost) отправляет значения удаляемой записи.

Метод действия Delete() для GET

Метод действия Delete() для GET функционирует точно так же, как метод действия Details():

```
[HttpGet("{id?}")]
public IActionResult Delete(int? id)
{
    var car = GetOneCar(id);
    if (car == null)
    {
        return NotFound();
    }
    return View(car);
}
```

Форму удаления можно просмотреть по ссылке /Cars/Delete/1 (рис. 31.9).

Delete Zippy

Are you sure you want to delete this car?

Make	VW
Color	Black
Pet Name	Zippy

[Delete](#) | [Back to List](#)

Рис. 31.9. Представление Delete

Метод действия Delete() для POST

Метод действия Delete() для POST просто отправляет значения Id и TimeStamp оболочке службы:

```
[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public IActionResult Delete(int id, Car car)
{
    if (id != car.Id)
    {
        return BadRequest();
    }
    _repo.Delete(car);
    return RedirectToAction(nameof(Index));
}
```

Метод действия Delete() для POST оптимизирован для отправки только значений, которые необходимы инфраструктуре EF Core для удаления записи.

На этом создание представлений и контроллера для сущности Car завершено.

Компоненты представлений

Компоненты представлений — еще одно новое функциональное средство, появившееся в ASP.NET Core. Они сочетают в себе преимущества частичных представлений и дочерних действий для визуализации частей пользовательского интерфейса. Как и частичные представления, компоненты представлений вызываются из другого представления, но в отличие от частичных представлений самих по себе компоненты представлений также имеют компонент серверной стороны. Благодаря такой комбинации они хорошо подходят для решения задач, подобных созданию динамических меню (как вскоре будет показано), панелей входа, содержимого боковой панели и всего того, что требует кода серверной стороны, но не может квалифицироваться как автономное представление.

На заметку! Дочерние действия в классической инфраструктуре ASP.NET MVC были методами действий контроллера, которые не могли служить конечными точками, видимыми клиенту. В ASP.NET Core они не существуют.

Для AutoLot компонент представления будет динамически создавать меню на основе производителей, которые присутствуют в базе данных. Меню отображается на каждой странице, поэтому вполне логичным местом для него является файл _Layout.cshtml. Но _Layout.cshtml не имеет компонента серверной стороны (в отличие от представлений), так что любое действие в приложении должно представлять данные компоновке _Layout.cshtml. Это можно делать в обработчике события OnActionExecuting() и в записях, помещаемых в объект ViewBag, но сопровождать подобное не будет простой задачей. Смешивание возможностей серверной стороны и инкапсуляции пользовательского интерфейса превращает такой сценарий в идеальный вариант для использования компонентов представлений.

Код серверной стороны

Создайте в корневом каталоге проекта AutoLot.Mvc новый каталог по имени ViewComponents и добавьте в него файл класса MenuViewComponent.cs. Подобно контроллерам классы компонентов представлений по соглашению именуются с суффиксом ViewComponent. И как у контроллеров, при обращении к компонентам представлений суффикс ViewComponent отбрасывается.

Добавьте в начало файла следующие операторы using:

```
using System.Linq;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
```

Сделайте класс общедоступным и унаследованным от ViewComponent. Компоненты представлений не обязательно наследовать от базового класса ViewComponent, но аналогично ситуации с базовым классом Controller наследование от ViewComponent упрощает большую часть работы. Создайте конструктор, который принимает экземпляр реализации интерфейса IMakeRepo и присваивает его переменной уровня класса. Пока что код выглядит так:

```
namespace AutoLot.Mvc.ViewComponents
{
    public class MenuViewComponent : ViewComponent
    {
        private readonly IMakeRepo _makeRepo;
        public MenuViewComponent(IMakeRepo makeRepo)
        {
            _makeRepo = makeRepo;
        }
    }
}
```

Компонентам представлений доступны два метода, Invoke() и InvokeAsync(). Один из них должен быть реализован и поскольку MakeRepo делает только синхронные вызовы, добавьте метод Invoke():

```
public async IViewComponentResult Invoke()
{
}
```

Когда компонент представления визуализируется из представления, вызывается открытый метод Invoke() / InvokeAsync(). Этот метод возвращает эк-

земпляр реализации интерфейса `IViewComponentResult`, который концептуально подобен `PartialViewResult`, но сильно упрощен. В методе `Invoke()` получается список производителей из хранилища и в случае успеха возвращается экземпляр `ContentViewComponentResult` (в его имени нет опечатки), где в качестве модели представления применяется список производителей. Если вызов для получения записей `Make` завершается неудачей, тогда производится возврат экземпляра `ContentViewComponentResult` с сообщением об ошибке. Модифицируйте код метода, как показано ниже:

```
public IViewComponentResult Invoke()
{
    var makes = _makeRepo.GetAll().ToList();
    if (!makes.Any())
    {
        return new ContentViewComponentResult("Unable to get the makes");
    }
    return View("MenuView", makes);
}
```

Вспомогательный метод `View()` из базового класса `ViewComponent` похож на вспомогательный метод с тем же именем из класса `Controller`, но с парой ключевых отличий. Первое отличие заключается в том, что стандартным именем файла представления является `Default.cshtml`, а не имя метода. Однако подобно вспомогательному методу `View()` из класса `Controller` имя представления может быть любым, когда оно передается вызову метода (без расширения `.cshtml`). Второе отличие связано с тем, что представление **обязано** находиться в одном из следующих трех каталогов:

```
Views/<имя_контроллера>/Components/<имя_компонента_представления>/
<имя_представления>
Views/Shared/Components/<имя_компонента_представления>/
<имя_представления>
Pages/Shared/Components/<имя_компонента_представления>/
<имя_представления>
```

На заметку! В версии ASP.NET Core 2.x появился еще один механизм для создания веб-приложений, который называется `Razor Pages`, но в этой книге он не рассматривается.

Класс C# может находиться где угодно (даже в другой сборке), но файл `<имя_представления>.cshtml` должен храниться в одном из ранее перечисленных каталогов.

Построение частичного представления

Частичное представление, визуализируемое классом `MenuViewComponent`, будет проходить по записям `Make`, добавляя каждую в виде элемента списка, который предназначен для отображения в меню Bootstrap. Элемент меню `All` (Все) добавляется первым как жестко закодированное значение.

Создайте внутри каталога `Views\Shared` новый каталог по имени `Components`, а в нем — еще один каталог под названием `Menu`. Имя каталога должно совпадать с именем созданного ранее класса компонента представления минус суффикс `ViewComponent`. Добавьте в каталог `Menu` файл частичного представления по имени `MenuView.cshtml`.

Удалите существующий код и поместите в файл показанную ниже разметку:

```
@model IEnumerable<Make>


All
@foreach (var item in Model)
{
    <a class="dropdown-item text-dark" asp-controller="Cars"
       asp-action="ByMake"
       asp-route-makeId="@item.Id"
       asp-route-makeName="@item.Name">@item.Name
}


```

Вызов компонентов представлений

Компоненты представлений обычно визуализируются из представления (хотя их можно визуализировать также из метода действия контроллера). Синтаксис довольно прямолинеен: `Component.Invoke(<имя_компонента_представления>)` или `@await Component.InvokeAsync(<имя_компонента_представления>)`. Как и в случае с контроллерами, при вызове компонента представления суффикс `ViewComponent` не должен указываться:

```
@await Component.InvokeAsync("Menu")      // асинхронная версия
@Component.Invoke("Menu")                  // синхронная версия
```

Вызов компонентов представлений как специальных вспомогательных функций дескрипторов

Появившиеся в ASP.NET 1.1 компоненты представлений можно вызывать с использованием синтаксиса вспомогательных функций дескрипторов. Вместо применения `Component.InvokeAsync()`/`Component.Invoke()` просто вызывайте компонент представления следующим образом:

```
<vc:menu></vc:menu>
```

В приложении потребуется разрешить использование такого способа вызова компонентов представлений, что делается добавлением команды `@addTagHelper` с именем сборки, которая содержит нужный компонент представления. В файл `_ViewImports.cshtml` необходимо добавить показанную ниже строку, которая уже была добавлена для специальных вспомогательных функций дескрипторов:

```
@addTagHelper *, AutoLot.Mvc
```

Обновление меню

Откройте частичное представление `_Menu.cshtml` и перейдите в место сразу после блока ``, который соответствует методу действия `Home/Index`. Поместите в частичное представление следующую разметку:

```
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle text-dark"
       data-toggle="dropdown">Inventory <i class="fa fa-car"></i></a>
    <vc:menu />
</li>
```

Строка, выделенная полужирным, визуализирует `MenuViewComponent` внутри меню. Окружающая ее разметка реализует форматирование Bootstrap.

Запустив приложение, вы увидите меню `Inventory` (Реестр), содержащее производителей в качестве элементов подменю (рис. 31.10).

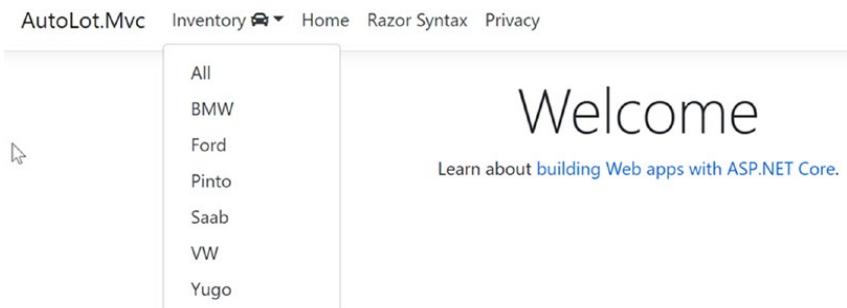


Рис. 31.10. Меню, обеспечиваемое компонентом представления `MenuViewComponent`

Пакетирование и минификация

При построении веб-приложений с применением библиотек клиентской стороны необходимо принять во внимание два дополнительных фактора, которые направлены на улучшение показателей производительности — пакетирование и минификацию.

Пакетирование

У веб-браузеров есть установленный предел на количество файлов, которые разрешено загружать параллельно из одной конечной точки. В случае использования с вашими файлами JavaScript и CSS приемов разработки SOLID, которые предусматривают разбиение связанного кода и стилей на более мелкие и управляемые файлы, могут возникать проблемы. Такой подход совершенствует процесс разработки, но становится причиной снижения производительности приложения из-за того, что файлы ожидают своей очереди на загрузку. Пакетирование — это просто объединение файлов с целью предотвращения их блокировки при достижении веб-браузером своего предела загрузки.

Минификация

Кроме того, для улучшения показателей производительности процесс минификации изменяет файлы CSS и JavaScript, уменьшая их размеры. Необязательные пробельные символы удаляются, а имена, не являющиеся ключевыми словами, делаются короче. Хотя файлы становятся практически нечитабельными для человека, функциональность не затрагивается, причем размеры файлов могут значительно сократиться. В свою очередь это ускоряет процесс загрузки, приводя к увеличению производительности приложения.

Решение `WebOptimizer`

Существует много инструментов разработки, которые позволяют пакетировать и минифицировать файлы как часть процесса сборки проекта. Безусловно, они эффективны, но могут стать проблематичными, если процессы перестают быть синхронизи-

рованными, поскольку на самом деле нет хорошего средства для сравнения исходных файлов с их пакетированными и минифицированными версиями.

WebOptimizer представляет собой пакет с открытым кодом, который обеспечивает пакетирование, минификацию и кеширование в качестве части конвейера ASP.NET Core. Он гарантирует, что пакетированные и минифицированные файлы соответствуют первоначальным файлам. Такие файлы не только точны, они еще и кешируются, значительно уменьшая количество операций дискового чтения для запросов страниц. Вы уже добавили пакет Libershark.WebOptimizer.Core при создании проектов в главе 29. Теперь пора им воспользоваться.

Обновление Startup.cs

Первый шаг предусматривает добавление WebOptimizer в конвейер. Откройте файл Startup.cs из проекта AutoLot.Mvc, отыщите в нем метод Configure() и добавьте в него следующую строку (сразу после вызова app.UseStaticFiles()):

```
app.UseWebOptimizer();
```

Следующим шагом будет конфигурирование того, что должно минифицироваться и пакетироваться. Обычно при разработке своего приложения вы хотите видеть непакетированные/неминифицированные версии файлов, но в подготовительной и производственной средах желательно применять пакетирование и минификацию. Добавьте показанный ниже блок кода в метод ConfigureServices():

```
if (_env.IsDevelopment() || _env.IsEnvironment("Local"))
{
    services.AddWebOptimizer(false, false);
}
else
{
    services.AddWebOptimizer(options =>
    {
        options.MinifyCssFiles(); // Минифицировать все файлы CSS.
        // options.MinifyJsFiles(); // Минифицировать все файлы JavaScript.
        options.MinifyJsFiles("js/site.js");
        options.MinifyJsFiles("lib/**/*.js");
    });
}
```

В случае среды Development пакетирование и минификация отключаются. Для остальных сред минифицируются все файлы CSS, файл site.js и все файлы JavaScript (с расширением .js) в каталоге lib и его подкаталогах. Обратите внимание, что все пути в проекте начинаются с каталога wwwroot.

WebOptimizer также поддерживает пакетирование. В первом примере создается пакет с использованием универсализации файловых имен, а во втором — пакет, для которого приводится список конкретных имен:

```
options.AddJavaScriptBundle("js/validations/validationCode.js",
    "js/validations/**/*.js");
options.AddJavaScriptBundle("js/validations/validationCode.js",
    "js/validations/validators.js", "js/validations/errorFormatting.js");
```

Важно отметить, что минифицированные и пакетированные файлы на самом деле не находятся на диске, а помещаются в кеш. Также важно отметить, что минифицированные файлы сохраняют то же самое имя (site.js) и не имеют обычное расширение .min(site.min.js).

На заметку! При обновлении своих представлений с целью добавления ссылок на пакетированные файлы среда Visual Studio сообщит о том, что они не существуют. Не переживайте, все будет визуализировано из кеша.

Обновление `_ViewImports.cshtml`

На финальном шаге в систему добавляются вспомогательные функции дескрипторов WebOptimizer. Они работают точно так же, как вспомогательные функции дескрипторов `asp-append-version`, описанные ранее в главе, но делают это автоматически для всех пакетированных и минифицированных файлов. Поместите в конец файла `_ViewImports.cshtml` следующую строку:

```
@addTagHelper *, WebOptimizer.Core
```

Шаблон параметров в ASP.NET Core

Шаблон параметров обеспечивает доступ сконфигурированных классов настроек к другим классам через внедрение зависимостей. Конфигурационные классы могут быть внедрены в другой класс с применением одной их версий `IOptions<T>`.

В табл. 31.6 кратко описан ряд версий интерфейса `IOptions`.

Таблица 31.6. Некоторые интерфейсы `IOptions`

Интерфейс	Описание
<code>IOptionsMonitor<T></code>	Извлекает параметры и поддерживает следующие возможности: уведомление об изменениях (с помощью <code>OnChange()</code>), перезагрузка конфигурации, именованные параметры (через <code>Get</code> и <code>CurrentValue</code>), а также выборочное объявление недействительными параметров
<code>IOptionsMonitorCache<T></code>	Кеширует экземпляры <code>T</code> с поддержкой полной/частичной недействительности/перезагрузки
<code>IOptionsSnapshot<T></code>	Рассчитывает параметры по каждому запросу
<code>IOptionsFactory<T></code>	Создает новые экземпляры <code>T</code>
<code>IOptions<T></code>	Корневой интерфейс. Не поддерживает <code>IOptionsMonitor<T></code> . Оставлен для обратной совместимости

Добавление информации об автодилере

На автомобильном сайте должна отображаться информация об автодилере, которая обязана быть настраиваемой без необходимости в повторном развертывании всего сайта, чего можно достичь с использованием шаблона параметров. Начните с добавления информации об автодилере в файл `appsettings.json`:

```
{
  "Logging": {
    "MSSqlServer": {
      "schema": "Logging",
      "tableName": "SeriLogs",
      "restrictedToMinimumLevel": "Warning"
    }
  }
}
```

```

        }
    },
    "ApplicationName": "AutoLot.MVC",
    "AllowedHosts": "*",
    "DealerInfo": {
        "DealerName": "Skimedic's Used Cars",
        "City": "West Chester",
        "State": "Ohio"
    }
}

```

Далее понадобится создать модель представления для хранения информации об автодилере. Добавьте в каталог Models проекта AutoLot.Mvc новый файл класса по имени DealerInfo.cs со следующим содержимым:

```

namespace AutoLot.Mvc.Models
{
    public class DealerInfo
    {
        public string DealerName { get; set; }
        public string City { get; set; }
        public string State { get; set; }
    }
}

```

На заметку! Конфигурируемый класс должен иметь открытый конструктор без параметров и не быть абстрактным. Стандартные значения можно устанавливать в свойствах класса.

Метод `Configure()` интерфейса `IServiceCollection` сопоставляет раздел конфигурационных файлов с конкретным типом. Затем этот тип может быть внедрен в классы и представления с применением шаблона параметров. Откройте файл `Startup.cs` и добавьте в него показанный ниже оператор `using`:

```
using AutoLot.Mvc.Models;
```

Перейдите к методу `ConfigureServices()` и поместите в него следующую строку кода:

```
services.Configure<DealerInfo>(Configuration.GetSection(nameof(DealerInfo)));
```

Откройте файл `HomeController.cs` и добавьте в него такой оператор `using`:

```
using Microsoft.Extensions.Options;
```

Затем модифицируйте метод `Index()`, как продемонстрировано далее:

```

[Route("")]
[Route("/{controller}")]
[Route("/{controller}/{action}")]
[HttpGet]
public IActionResult Index([FromServices] IOptionsMonitor<DealerInfo>
dealerMonitor)
{
    var vm = dealerMonitor.CurrentValue;
    return View(vm);
}

```

Когда класс сконфигурирован в коллекции служб и добавлен в контейнер DI, его можно извлечь с использованием шаблона параметров. В рассматриваемом примере OptionsMonitor будет читать конфигурационный файл, чтобы создать экземпляр класса DealerInfo. Свойство CurrentValue получает экземпляр DealerInfo, созданный из текущего файла настроек (даже если файл изменился после запуска приложения). Затем экземпляр DealerInfo передается представлению Index.cshtml.

Обновите представление Index.cshtml, расположенное в каталоге Views\Home, чтобы оно было строго типизированным для класса DealerInfo и отображало свойства модели:

```
@model AutoLot.Mvc.Models.DealerInfo
{
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">Welcome to @Model.DealerName</h1>
    <p class="lead">Located in @Model.City, @Model.State</p>
</div>
```

На заметку! За дополнительными сведениями о шаблоне параметров в ASP.NET Core обращайтесь в документацию по ссылке <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/configuration/options>.

Создание оболочки службы

Вплоть до этого момента в приложении AutoLot.Mvc применялся уровень доступа к данным напрямую. Еще один подход предусматривает использование службы AutoLot.Api, позволяя ей обрабатывать весь доступ к данным.

Обновление конфигурации приложения

Конечные точки приложения AutoLot.Api будут варьироваться на основе среды. Скажем, при разработке на вашей рабочей станции базовый URI выглядит как <https://localhost:5021>. В промежуточной среде им может быть <https://mytestserver.com>. Осведомленность о среде в сочетании с обновленной конфигурационной системой (представленной в главе 29) будут применяться для добавления разных значений.

Файл appsettings.Development.json добавит информацию о службе для локальной машины. По мере того как код перемещается по разным средам, настройки будут обновляться в специфическом файле среды, чтобы соответствовать базовому URI и конечным точкам для этой среды. В рассматриваемом примере вы обновляете только настройки для среды Development. Откройте файл appsettings.Development.json и модифицируйте его следующим образом (изменения выделены полужирным):

```
{
    "Logging": {
        "MSSqlServer": {
            "schema": "Logging",
            "tableName": "SeriLogs",
            "restrictedToMinimumLevel": "Warning"
        }
    },
}
```

```

    "RebuildDataBase": false,
    "ApplicationName": "AutoLot.Mvc - Dev",
    "ConnectionStrings": {
        "AutoLot":
            "Server=.,5433;Database=AutoLot;User ID=sa;Password=P@ssw0rd;"}
    },
    "ApiServiceSettings": {
        "Uri": "https://localhost:5021/",
        "CarBaseUri": "api/Cars",
        "MakeBaseUri": "api/Makes"
    }
}
}

```

На заметку! Удостоверьтесь, что номер порта соответствует вашей конфигурации для AutoLot.Api.

За счет использования конфигурационной системы ASP.NET Core и обновления файлов, специфичных для среды (например, appsettings.staging.json и appsettings.production.json), ваше приложение будет располагать надлежащими значениями без необходимости в изменении кода.

Создание класса ApiServiceSettings

Настройки службы будут заполняться из настроек таким же способом, как и информация об автодилере. Создайте в проекте AutoLot.Services новый каталог по имени ApiWrapper и добавьте в него файл класса ApiServiceSettings.cs. Имена свойств класса должны совпадать с именами свойств в разделе ApiServiceSettings файла appsettings.Development.json. Код класса показан ниже:

```

namespace AutoLot.Services.ApiWrapper
{
    public class ApiServiceSettings
    {
        public ApiServiceSettings() { }
        public string Uri { get; set; }
        public string CarBaseUri { get; set; }
        public string MakeBaseUri { get; set; }
    }
}

```

Оболочка службы API

В версии ASP.NET Core 2.1 появился интерфейс IHTTPClientFactory, который позволяет конфигурировать строго типизированные классы для вызова внутри служб REST. Создание строго типизированного класса дает возможность инкапсулировать все обращения к API в одном месте. Это централизует взаимодействие со службой, конфигурацию клиента HTTP, обработку ошибок и т.д. Затем класс можно добавить в контейнер DI для дальнейшего применения в приложении. Контейнер DI и реализация IHTTPClientFactory обрабатывают создание и освобождение HttpClient.

Интерфейс IApiServiceWrapper

Интерфейс оболочки службы AutoLot содержит методы для обращения к службе AutoLot.Api. Создайте в каталоге ApiWrapper новый файл интерфейса IApiServiceWrapper.cs и приведите операторы using к следующему виду:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using AutoLot.Models.Entities;
```

Модифицируйте код интерфейса, как показано ниже:

```
namespace AutoLot.Services.ApiWrapper
{
    public interface IApiServiceWrapper
    {
        Task<IList<Car>> GetCarsAsync();
        Task<IList<Car>> GetCarsByMakeAsync(int id);
        Task<Car> GetCarAsync(int id);
        Task<Car> AddCarAsync(Car entity);
        Task<Car> UpdateCarAsync(int id, Car entity);
        Task DeleteCarAsync(int id, Car entity);
        Task<IList<Make>> GetMakesAsync();
    }
}
```

Класс ApiServiceWrapper

Создайте в каталоге ApiWrapper проекта AutoLot.Services новый файл класса по имени ApiServiceWrapper.cs и модифицируйте его операторы using следующим образом:

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Json;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;
using AutoLot.Models.Entities;
using Microsoft.Extensions.Options;
```

Сделайте класс открытым и добавьте конструктор, который принимает экземпляр HttpClient и экземпляр реализации IOptionsMonitor<ApiServiceSettings>. Создайте закрытую переменную типа ServiceSettings и присвойте ей значение с использованием свойства CurrentValue параметра IOptionsMonitor<Service Settings>. Код показан ниже:

```
public class ApiServiceWrapper : IApiServiceWrapper
{
    private readonly HttpClient _client;
    private readonly ApiServiceSettings _settings;
    public ApiServiceWrapper(HttpClient client,
        IOptionsMonitor<ApiServiceSettings> settings)
    {
        _settings = settings.CurrentValue;
        _client = client;
        _client/BaseAddress = new Uri(_settins.Uri);
    }
}
```

На заметку! В последующих разделах содержится много кода без какой-либо обработки ошибок. Поступать так настоятельно не рекомендуется! Обработка ошибок здесь опущена из-за экономии пространства.

Внутренние поддерживающие методы

Класс содержит четыре поддерживающих метода, которые применяются открытыми методами.

Вспомогательные методы для POST и PUT

Следующие методы являются оболочками для связанных методов HttpClient:

```
internal async Task<HttpResponseMessage> PostAsJson(string uri,
                                                       string json)
{
    return await _client.PostAsync(uri, new StringContent(json, Encoding.UTF8,
                                                            "application/json"));
}

internal async Task<HttpResponseMessage> PutAsJson(string uri, string json)
{
    return await _client.PutAsync(uri, new StringContent(json, Encoding.UTF8,
                                                            "application/json"));
}
```

Вспомогательный метод для DELETE

Последний вспомогательный метод используется для выполнения HTTP-метода DELETE. Спецификация HTTP 1.1 (и более поздние версии) позволяет передавать тело в HTTP-методе DELETE, но для этого пока еще не предусмотрено расширяющего метода HttpClient. Экземпляр HttpRequestMessage потребуется создавать с нуля.

Первым делом необходимо создать сообщение запроса с применением инициализации объектов для установки Content, Method и RequestUri. Затем сообщение отправляется, после чего ответ возвращается вызывающему коду. Вот код метода:

```
internal async Task<HttpResponseMessage> DeleteAsJson(string uri,
                                                       string json)
{
    HttpRequestMessage request = new HttpRequestMessage
    {
        Content = new StringContent(json, Encoding.UTF8, "application/json"),
        Method = HttpMethod.Delete,
        RequestUri = new Uri(uri)
    };
    return await _client.SendAsync(request);
}
```

Вызовы HTTP-метода GET

Есть четыре вызова HTTP-метода GET: один для получения всех записей Car, один для получения записей Car по производителю Make, один для получения одиночной записи Car и один для получения всех записей Make. Все они следуют тому же самому шаблону. Метод GetAsync() вызывается для возвращения экземпляра HttpResponseMessage. Успешность или неудача вызова проверяется с помощью метода EnsureSuccessStatusCode(), который генерирует исключение, если вызов не

возвратил код состояния успеха. Затем тело ответа сериализируется в тип свойства (сущность или список сущностей) и возвращается вызывающему коду. Ниже приведен код всех методов:

```
public async Task<IList<Car>> GetCarsAsync()
{
    var response =
        await _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}");
    response.EnsureSuccessStatusCode();
    var result = await response.Content.ReadFromJsonAsync<IList<Car>>();
    return result;
}

public async Task<IList<Car>> GetCarsByMakeAsync(int id)
{
    var response = await
        _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}/bymake/{id}");
    response.EnsureSuccessStatusCode();
    var result = await response.Content.ReadFromJsonAsync<IList<Car>>();
    return result;
}

public async Task<Car> GetCarAsync(int id)
{
    var response = await
        _client.GetAsync($"{_settings.Uri}{_settings.CarBaseUri}/{id}");
    response.EnsureSuccessStatusCode();
    var result = await response.Content.ReadFromJsonAsync<Car>();
    return result;
}

public async Task<IList<Make>> GetMakesAsync()
{
    var response =
        await _client.GetAsync($"{_settings.Uri}{_settings.MakeBaseUri}");
    response.EnsureSuccessStatusCode();
    var result = await response.Content.ReadFromJsonAsync<IList<Make>>();
    return result;
}
```

Вызов HTTP-метода POST

Метод для добавления записи Car использует HTTP-метод POST. Он применяет вспомогательный метод для отправки сущности в формате JSON и возвращает запись Car из тела ответа. Вот его код:

```
public async Task<Car> AddCarAsync(Car entity)
{
    var response = await PostAsJson($"{_settings.Uri}{_settings.CarBaseUri}",
        JsonSerializer.Serialize(entity));
    if (response == null)
    {
        throw new Exception("Unable to communicate with the service");
    }
    return await response.Content.ReadFromJsonAsync<Car>();
}
```

Вызов HTTP-метода PUT

Метод для обновления записи Car использует HTTP-метод PUT. Он применяет вспомогательный метод для отправки записи Car в формате JSON и возвращает обновленную запись Car из тела ответа:

```
public async Task<Car> UpdateCarAsync(int id, Car entity)
{
    var response =
        await PutAsJson(${_settings.Uri}{_settings.CarBaseUri}/{id}",
                        JsonSerializer.Serialize(entity));
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadFromJsonAsync<Car>();
}
```

Вызов HTTP-метода DELETE

Последний добавляемый метод предназначен для выполнения HTTP-метода DELETE. Шаблон соответствует остальным методам: использование вспомогательного метода и проверка ответа на предмет успешности. Он ничего не возвращает вызывающему коду, поскольку сущность была удалена. Ниже показан код метода:

```
public async Task DeleteCarAsync(int id, Car entity)
{
    var response =
        await DeleteAsJson(${_settings.Uri}{_settings.CarBaseUri}/{id}",
                        JsonSerializer.Serialize(entity));
    response.EnsureSuccessStatusCode();
}
```

Конфигурирование служб

Создайте в каталоге ApiWrapper проекта AutoLot.Service новый файл класса по имени ServiceConfiguration.cs. Приведите операторы using к следующему виду:

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

Сделайте класс открытым и статическим, после чего добавьте открытый статический расширяющий метод для IServiceCollection:

```
namespace AutoLot.Services.ApiWrapper
{
    public static class ServiceConfiguration
    {
        public static IServiceCollection Configure ApiServiceWrapper(
            this IServiceCollection services, IConfiguration config)
        {
            return services;
        }
    }
}
```

В первой строке расширяющего метода в контейнер DI добавляется ApiServiceSettings. Во второй строке в контейнер DI добавляется IApiServiceWrapper и регистрируется класс с помощью фабрики HttpClient. Это позволяет

внедрять IApiServiceWrapper в другие классы, а фабрика HttpClient будет управлять внедрением и временем существования HttpClient:

```
public static IServiceCollection ConfigureApiServiceWrapper(
    this IServiceCollection services, IConfiguration config)
{
    services.Configure<ApiServiceSettings>(config.GetSection(
        nameof(ApiServiceSettings)));
    services.AddHttpClient<IApiServiceWrapper, ApiServiceWrapper>();
    return services;
}
```

Откройте файл Startup.cs и добавьте следующий оператор using:

```
using AutoLot.Services.ApiWrapper;
```

Перейдите к методу ConfigureServices() и добавьте в него показанную ниже строку:

```
services.ConfigureApiServiceWrapper(Configuration);
```

Построение класса CarsController

Текущая версия CarsController жестко привязана к хранилищам в библиотеке доступа к данным. Следующая итерация CarsController для связи с базой данных будет применять оболочку службы. Переименуйте CarsController в CarsDalController (включая конструктор) и добавьте в каталог Controllers новый класс по имени CarsController. Код этого класса является практически точной копией CarsController, но они хранятся по отдельности с целью прояснения разницы между использованием хранилищ и службы.

На заметку! При работе с одной и той же базой данных вам редко придется применять вместе уровень доступа к данным и оболочку службы. Здесь показаны оба варианта, чтобы вы смогли решить, какой из них лучше подходит в вашей ситуации.

Приведите операторы using к следующему виду:

```
using System.Threading.Tasks;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Models.Entities;
using AutoLot.Services.ApiWrapper;
using AutoLot.Services.Logging;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
```

Далее сделайте класс открытым, унаследуйте его от Controller и добавьте атрибут Route. Создайте конструктор, который принимает экземпляры реализаций IAutoLotServiceWrapper и IAppLogging, после чего присвойте оба экземпляра переменным уровня класса. Вот начальный код:

```
namespace AutoLot.Mvc.Controllers
{
    [Route("[controller]/[action]")]
    public class CarsController : Controller
    {
```

```
private readonly IApiServiceWrapper _serviceWrapper;
private readonly IAppLogging<CarsController> _logging;
public CarsController(IApiServiceWrapper serviceWrapper,
    IAppLogging<CarsController> logging)
{
    _serviceWrapper = serviceWrapper;
    _logging = logging;
}
}
```

Вспомогательный метод GetMakes()

Вспомогательный метод GetMakes() строит экземпляр SelectList со всеми записями Make в базе данных. Он использует Id в качестве значения и Name в качестве отображаемого текста:

```
internal async Task<SelectList> GetMakesAsync()=>
    new SelectList(
        await _serviceWrapper.GetMakesAsync(),
        nameof(Make.Id),
        nameof(Make.Name));
```

Вспомогательный метод GetOneCar ()

Вспомогательный метод GetOneCar() получает одиночную запись Car:

```
internal async Task<Car> GetOneCarAsync(int? id)
=> !id.HasValue ? null : await serviceWrapper.GetCarAsync(id.Value);
```

Открытые методы действий

Единственное отличие между открытыми методами действий в этом контроллере и аналогичными методами в CarsDalController связано с доступом к данным, а также с тем, что все методы определены как асинхронные. Поскольку вы уже понимаете, для чего предназначено то или иное действие, ниже приведены остальные методы, изменения в которых выделены полужирным:

```
[Route("/{controller}")]  
[Route("/{controller}/{action}")]  
public async Task<IActionResult> Index()  
    => View(await _serviceWrapper.GetCarsAsync());  
  
[HttpGet("{makeId}/{makeName}")]  
public async Task<IActionResult> ByMake(int makeId, string makeName)  
{  
    ViewBag.MakeName = makeName;  
    return View(await _serviceWrapper.GetCarsByMakeAsync(makeId));  
}  
  
[HttpGet("{id?}")]
public async Task<IActionResult> Details(int? id)
{
    if (!id.HasValue)
    {
        return BadRequest();
    }
}
```

```
var car = await GetOneCarAsync(id);
if (car == null)
{
    return NotFound();
}
return View(car);
}

[HttpGet]
public async Task<IActionResult> Create()
{
    ViewData["MakeId"] = await GetMakesAsync();
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(Car car)
{
    if (ModelState.IsValid)
    {
        await _serviceWrapper.AddCarAsync(car);
        return RedirectToAction(nameof(Index));
    }
    ViewData["MakeId"] = await GetMakesAsync();
    return View(car);
}

[HttpGet("{id?}")]
public async Task<IActionResult> Edit(int? id)
{
    var car = await GetOneCarAsync(id);
    if (car == null)
    {
        return NotFound();
    }
    ViewData["MakeId"] = await GetMakesAsync();
    return View(car);
}

[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, Car car)
{
    if (id != car.Id)
    {
        return BadRequest();
    }
    if (ModelState.IsValid)
    {
        await _serviceWrapper.UpdateCarAsync(id, car);
        return RedirectToAction(nameof(Index));
    }
    ViewData["MakeId"] = await GetMakesAsync();
    return View(car);
}
```

```
[HttpGet("{id?}")]
public async Task<IActionResult> Delete(int? id)
{
    var car = await GetOneCarAsync(id);
    if (car == null)
    {
        return NotFound();
    }
    return View(car);
}

[HttpPost("{id}")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, Car car)
{
    await _serviceWrapper.DeleteCarAsync(id, car);
    return RedirectToAction(nameof(Index));
}
```

Обновление компонента представления

В текущий момент внутри компонента представления MenuViewComponent применяется уровень доступа к данным и синхронная версия `Invoke()`. Внесите в класс следующие изменения:

```
using System.Linq;
using System.Threading.Tasks;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Services.ApiWrapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AutoLot.Mvc.ViewComponents
{
    public class MenuViewComponent : ViewComponent
    {
        private readonly IApiServiceWrapper _serviceWrapper;
        public MenuViewComponent(IApiServiceWrapper serviceWrapper)
        {
            _serviceWrapper = serviceWrapper;
        }

        public async Task<IViewComponentResult> InvokeAsync()
        {
            var makes = await _serviceWrapper.GetMakesAsync();
            if (makes == null)
            {
                return new ContentViewComponentResult("Unable to get the makes");
            }
            return View("MenuView", makes);
        }
    }
}
```

Совместный запуск приложений AutoLot.Mvc и AutoLot.Api

Приложение AutoLot.Mvc рассчитывает на то, что приложение AutoLot.Api должно быть запущено. Это можно сделать с помощью Visual Studio, командной строки или через комбинацию того и другого.

На заметку! Вспомните, что приложения AutoLot.Mvc и AutoLot.Api сконфигурированы на воссоздание базы данных при каждом их запуске. Обязательно отключите воссоздание хотя бы в одном из приложений, иначе возникнет конфликт. Чтобы ускорить отладку, отключите воссоздание в обоих приложений при тестировании функциональности, которая не изменяет данные.

Использование Visual Studio

Вы можете сконфигурировать среду Visual Studio на запуск нескольких проектов одновременно. Щелкните правой кнопкой мыши на имени решения в окне Solution Explorer, выберите в контекстном меню пункт Select Startup Projects (Выбрать стартовые проекты) и установите действия для проектов AutoLot.Api и AutoLot.Mvc в Start (Запуск), как показано на рис. 31.11.

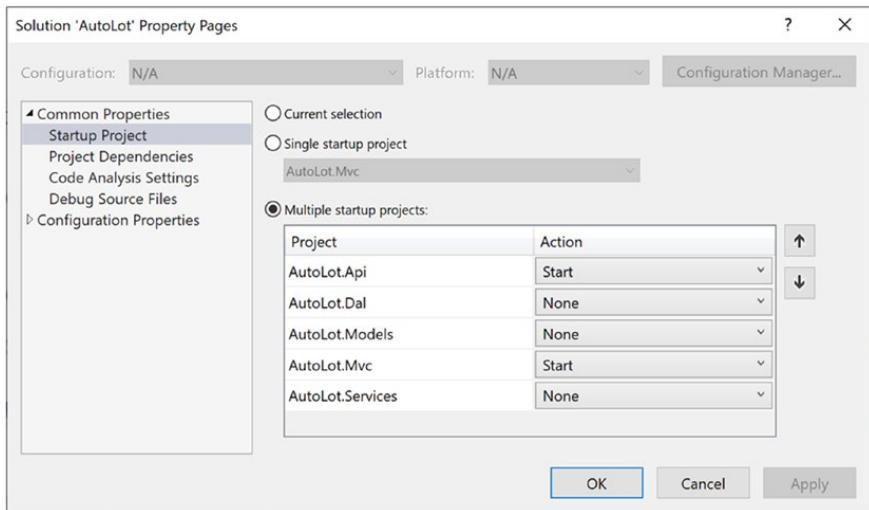


Рис. 31.11. Установка нескольких стартовых проектов в Visual Studio

После нажатия клавиши <F5> (или щелчка на кнопке запуска с зеленой стрелкой) оба проекта запустятся. При этом возникает ряд сложностей. Первая сложность — среда Visual Studio запоминает последний профиль, который применялся для запуска приложения. Это значит, что если вы использовали для запуска AutoLot.Api веб-сервер IIS Express, то запуск обоих приложений приведет к запуску AutoLot.Api с применением IIS Express, поэтому порт в настройках служб окажется некорректным.

Проблему легко устраниТЬ. Либо измените порты в файле `appsettings.development.json`, либо запустите приложение под управлением Kestrel, прежде чем конфигурировать совместный запуск приложений.

Вторая сложность связана с синхронизацией. Оба проекта стартуют практически одновременно. Если вы сконфигурировали приложение `AutoLot.Api` на воссоздание базы данных при каждом его запуске, тогда она не будет готова для приложения `AutoLot.Mvc`, когда компонент представления запускается с целью построения меню. Проблему решит быстрое обновление браузера, отображающего `AutoLot.Mvc` (как только вы увидите пользовательский интерфейс Swagger в `AutoLot.Api`).

Использование командной строки

Откройте окно командной строки в каждом каталоге проекта и введите команду `dotnet watch run`. Это позволит управлять порядком и синхронизацией, а также гарантирует, что приложения выполняются с применением Kestrel, но не IIS. Информацию об отладке при запуске из командной строки ищите в главе 29.

Резюме

В настоящей главе вы завершили изучение ASP.NET Core, равно как и построение приложения `AutoLot.Mvc`. Процесс изучения начался с исследования представлений, частичных представлений, а также шаблонов редактирования и отображения. Затем вы узнали о вспомогательных функциях дескрипторов, смешивающих разметку клиентской стороны с кодом серверной стороны.

Следующие темы касались библиотек клиентской стороны, включая управление библиотеками в проекте плюс пакетирование и минификацию. После конфигурирования компоновка была обновлена с учетом новых путей к библиотекам и разбита на набор частичных представлений, а с целью дальнейшей детализации обработки клиентских библиотек была добавлена вспомогательная функция дескриптора для среды.

Затем с использованием `HttpClientFactory` и конфигурационной системы ASP.NET Core была создана оболочка службы, взаимодействующая с `AutoLot.Api`, которая применялась для создания компонента представления, отвечающего за построение динамической системы меню. После краткого обсуждения способов одновременной загрузки обоих приложений (`AutoLot.Api` и `AutoLot.Mvc`) была разработана основная часть приложения.

Разработка начиналась с создания контроллера `CarsController` и всех методов действий. Далее были добавлены специальные вспомогательные функции дескрипторов и в заключение созданы все представления, касающиеся записей `Car`. Конечно, был построен только один контроллер и его представления, но с помощью продемонстрированного шаблона можно создать контроллеры и представления для всех существующих `AutoLot`.

Предметный указатель

A

Active Data Objects (ADO), 62
AppLogging, 519
ASP.NET Core, 479; 555
Azure Data Studio, 74; 78

B

BAML (Binary Application Markup Language), 294

C

Command-Line Interface (CLI), 127
Create, Read, Update, Delete (CRUD), 99; 128; 183
Cross-Origin Requests (CORS), 553
CSS, 594

D

Data Access Layer (DAL), 440
Dependency Injection (DI), 496
Document-Type Definition (DTD), 49

E

EDM (Entity data model), 137
EF (Entity Framework), 126
Entity Data Model (EDM),
Entity Framework (EF),
XAML (Extensible Application Markup Language), 266

H

HTML helper, 560
HTTP (Hypertext Transfer Protocol), 530; 590
HTTP Strict Transport Security (HSTS), 515

I

Internet Information Services (IIS), 479

J

JavaScript, 594
JSON (JavaScript Object Notation), 45

K

Kaxaml, 278; 279

M

MVC (Model-View-Controller), 478; 555
MVV (MModel View ViewModel), 439

N

NuGet, 69; 184; 502

O

ORM (Object-relational mapping), 61; 99; 128
OpenAPI, 536

R

Razor, 558
Синтаксис Razor, 569
REST (REpresentational State Transfer), 530

S

SQL Server Management Studio (SSMS), 74
Swagger, 536; 539

T

Table-per-hierarchy (TPH), 138
Table-per-type (TPT), 138
Tag helper, 570

V

Visual Studio, 317; 376; 382; 500; 505; 507; 592; 625

W

WebOptimizer, 612
WCF (Windows Communication Foundation), 480
WPF (Windows Presentation Foundation), 301

X

XML (eXtensible Markup Language), 45

A

Анимация, 411
данных, 151
Атрибуты проверки достоверности, 488

Б

База данных
добавление представления базы
данных, 204
Библиотека
диспетчер библиотек, 592
клиентской стороны, 591; 592

В

Ввод, 578
Ввод-вывод
файловый, 20
Внедрение зависимостей, 496

Г

Граф
добавление объектного графа, 258
объектов, 46

Д

Данные
адаптеры данных, 67
аннотации данных, 151
добавление данных, 443
поставщики данных, 62
привязка данных, 344
связанные, 163
фильтрация связанных
данных, 247
Действия, 482; 484
Десериализация, 49; 53
Диспетчер библиотек, 592

З

Загрузка
ленивая, 166
энергичная, 164
явная, 165
Запись
создание записей, 255
добавление нескольких записей
одновременно, 257
обновление записей, 259
удаление записей, 262
Запрос
SQL
параметризованный, 106
выполнение запросов SQL с помощью
LINQ, 250
глобальные фильтры запросов, 242
между источниками (CORS), 553
типы запросов, 136

И

Изображение, 584
Инструмент Kaxaml, 278
Интерфейс, 613
IDbCommand, 66
IDbConnection, 65
IDbDataParameter, 67
IDbTransaction, 66
IDisposable, 101
IRepo, 208
XAML, 293
Инфраструктура
ADO.NET, 61; 126
EF Core, 166
Prism, 441
WPF, 301
Исключение
специальное, 200
тестирование фильтра
исключений, 553

К

Класс
AddCarCommand, 469
ApiServiceSettings, 616
ApiServiceWrapper, 617
Application, 272
ApplicationDbContext, 197
BaseEntity, 189
BaseRepo, 208; 210
Brush, 375
CarsController, 547; 596; 621
ChangeColorCommand, 466; 470
CommandBase, 468
ContentControl, 274
ContentPresenter, 436
Control, 275
Controller, 482
 ControllerBase, 483
DbContext, 129; 132
DbSet<T>, 134
DependencyObject, 277
Directory, 22
DispatcherObject, 278
Drawing, 386
DrawingImage, 386
DrawingVisual, 392; 397
File, 22
FileInfo, 28
FileStream, 36
FileSystemInfo, 22
FrameworkElement, 276
Geometry, 372

InventoryDal, 100; 105
 JsonSerializer, 53
 MigrationHelpers, 204
 ObservableCollection<T>, 447
 Path, 372
 Person, 189
 RelayCommand, 470
 ScrollViewer, 316
 Shape, 365; 367
 SqlBulkCopy, 117
 Startup, 489; 509; 536
 Stream, 35
 StreamReader, 37
 StreamWriter, 37; 40
 StringWriter, 40
 Timeline, 413
 Transform, 379
 UIElement, 277
 Validation, 450
 ViewResult, 555
 Visual, 277; 392
 Window, 273
 XmlSerializer, 49; 50
 создание базового класса, 585

Ключевые
 кадры, 418
 слова XAML, 281
Командная строка, 592; 626
Компоновка, 565
 стандартная
 для представлений, 566
Конвейер обработки, 552
Конструктор, 510
 добавление конструкторов, 100
Контроллер, 479; 482
 обновление контроллера, 526
Конфигурация приложений, 499
Конфигурирование
 настройки запуска, 505
 служб, 620

Л

Ленивая загрузка, 166

М

Маркер, 489
Маршрут
 маркеры маршрутов, 489
 маршрутизация, 489
 на основе соглашений, 489
Методы
 агрегирования, 251
Механизм визуализации Razor, 558
Миграция
 обновление миграции, 205

применение миграции, 205
Минифициация, 611
Модель, 99; 440; 478
 неявная привязка моделей, 485
 представления, 440
 привязка моделей, 484
 сущностных данных, 137

Н

Навигационные свойства, 141

О

Обработка событий, 202
Объект
 серIALIZАЦИЯ
 коллекций объектов, 52
 объектов, 20
 подключений, 92
 чтения данных, 97
Оператор
 LINQ, 168
 пакетирование операторов, 169
Отладчик XAML, 292

Отношения
 “многие ко многим”, 146
 “один к одному”, 144
 “один ко многим”, 142
Ошибка
 достоверности
 контроль ошибок проверки
 достоверности, 461
 на основе аннотаций данных, 461
 отображение с помощью стандартной
 страницы ошибок, 558
 отображение с помощью страницы
 исключений для разработчиков, 557
 сообщение об ошибке, 557

П

Пакет
 NuGet, 69; 184; 502
 WebOptimizer, 612
Пакетирование, 611
Параллелизм
 проверка параллелизма, 261
Параметры-заполнители, 106
Паттерн
 MVVM, 439
 “модель-представление-контроллер”
 (MVC), 478
Перо, 378
Позиционирование, 35
Поле
 включение полей, 54

Поставщики данных

ADO.NET, 64

абстрагирование поставщиков
данных с использованием
интерфейсов, 69

Поток

закрытие потока, 35

освобождение потока, 35

Представление, 440; 479; 561

компоненты представлений, 607

модели представлений, 568

создание частичных
представлений, 567

строго типизированные, 568

частичное, 609

частичные, 566

Привязка

добавление привязок, 443

данных, 344

моделей, 484

неявная, 485

явная, 486

Приложение

клиентское

создание консольного клиентского
приложения, 111

конфигурация приложений, 499

построение класса приложения, 272

Проверка достоверности, 580

Пространство

имен XML, 280

имен System.Data, 64

имен System.IO, 20

P

Разделители, 314

Раскладовка, 417

Редактор

Kaxaml, 279

Visual Studio, 376

XAML, 291

трансформаций, 384

Visual Studio, 382

Ресурс

двоичный, 400

логический, 399

объединенный словарь ресурсов, 409

объектный, 399

C

Сборки WPF, 270

Свойство

CLR, 352

зависимости, 277; 414

зависимости, 349; 352; 353

навигационные, 141; 167

Сериализация, 49; 53

коллекций объектов, 20; 45; 52; 59

Синтаксис Razor, 569

Словарь ресурсов, 409

ModelState, 485

Служба

конфигурирование служб, 620

сериализации объектов, 45

Событие

маршрутизуемые, 330

обработка событий, 202

прямое, 332

пузырьковые, 331; 333

триггеры событий, 418

туннельные, 331; 333

Соглашение

об именовании, 481

об ошибке

отображение с помощью стандартной
страницы ошибок, 558

отображение с помощью страницы
исключений для разработчиков, 557
по конфигурации, 481

Среда, 581

Ссылка, 582

Стили с анимацией, 424

Строка подключения, 92

Сущность 99; 126; 137

Car (Inventory), 190

Car, 199

CreditRisk, 194; 198

Customer, 192; 198

Make, 193; 199

Order, 195; 199

SeriLogEntry, 196; 197

интерфейсы хранилищ, специфичных
для сущностей, 212

обновление

неотслеживаемых сущностей, 260

отслеживаемых сущностей, 259

состояние сущности, 255

удаление

неотслеживаемых сущностей, 263

отслеживаемых сущностей, 263

Сценарий, 582

T

Таблица маршрутов, 489

Текстовая область, 579

Теория, 230

Типы запросов, 136

Транзакция, 112; 133

Трансформация

графическая, 378

Триггер, 418; 424

событий, 418

У**Установка**

IDE-среды SQL Server, 74
SQL Server 2019, 74

Утилита

msbuild.exe, 294

Ф**Фабрика DbContext**, 131**Файл**

JavaScript, 594

libman.json, 592

SVG, 389

XAML, 389

текстовый, 38

Файловый ввод-вывод, 20**Факты**, 230**Фильтрация связанных данных**, 247**Фильтры**, 495

авторизации, 495

действий, 496

добавление фильтров в конвейер обработки, 552

исключений, 496; 550

результатов, 496

ресурсов, 495

запросов

глобальные, 166; 242

Форма, 576**Формат BAML**, 296**Функция**

DisplayFor(), 591

DisplayForModel(), 591

EditorFor(), 591

EditorForModel(), 591

HTML

вспомогательные, 560; 590

дескриптора (tag helper)

вспомогательная для, 570

вывода сведений об элементе, 586

действия формы, 577

изображения, 584

проверки достоверности, 580

редактирования сведений об

элементе, 588

специальная, 584

среды, 581

ссылки, 582

сценария, 582

текстовой области, 579

удаления элемента, 587

формы, 576

элемента ввода, 578

элемента выбора, 579

якоря, 578

Х**Хранилище данных**, 214**Хранимая процедура**, 109; 204

получение данных из хранимых процедур, 254

Ш**Шаблон**, 433

отображения, 563

параметров, 613

Штрих, 340**Э****Элемент**

ввода, 578

выбора, 579

управления WPF, 303

Я**Явная загрузка**, 165**Язык**

XAML, 266; 278

атрибуты XAML, 283

ключевые слова XAML, 281

расширение разметки XAML, 286

элементы XAML, 283

XML, 45

Якорь, 578

У 10-му виданні книги описані новітні можливості мови C# 9 та .NET 5 разом із докладним “закулісним” обговоренням, покликаним розширити навички критичного мислення розробників, коли йдеться про їхнє ремесло. Книга охоплює ASP.NET Core, Entity Framework Core та багато іншого поряд з останніми оновленнями уніфікованої платформи. Усі приклади коду було переписано з урахуванням можливостей останнього випуску C#9.

Науково-популярне видання
Троєлсен, Ендрю, Джепікс, Філіп

**Мова програмування C# 9 і платформа .NET 5:
основні принципи та практики програмування
Том 2, 10-е видання**
(Рос. мовою)

Із загальних питань звертайтеся до видавництва “Діалектика” за адресою:
info@dialektika.com, <http://www.dialektika.com>

Підписано до друку 11.02.2022. Формат 60x90/16
Ум. друк. арк. 39,5. Обл.-вид. арк. 36,2

Видавець ТОВ “Комп’ютерне видавництво “Діалектика”
03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.
Свідоцтво суб’єкта видавничої справи ДК № 6758 від 16.05.2019.

ПРОФЕССИОНАЛАМ ОТ ПРОФЕССИОНАЛОВ

Язык программирования C# 9 и платформа .NET 5: основные принципы и практики программирования

2
ТОМ

Эта классическая книга представляет собой всеобъемлющий источник сведений о языке программирования C# и о связанной с ним инфраструктуре. В 10-м издании книги вы найдете описание новейших возможностей языка C# 9 и .NET 5 вместе с подробным "закулисным" обсуждением, призванным расширить навыки критического мышления разработчиков, когда речь идет об их ремесле. Книга охватывает ASP.NET Core, Entity Framework Core и многое другое наряду с последними обновлениями унифицированной платформы .NET, начиная с улучшений показателей производительности настольных приложений Windows в .NET 5 и обновления инструментария XAML и заканчивая расширенным рассмотрением файлов данных и способов обработки данных. Все примеры кода были переписаны с учетом возможностей последнего выпуска C# 9.

Погрузитесь в книгу и выясняйте, почему она является лидером у разработчиков по всему миру. Сформируйте прочный фундамент в виде знания приемов объектно-ориентированного проектирования, атрибутов и рефлексии, обобщений и коллекций, а также множества более сложных тем, которые не раскрываются в других книгах (коды операций CIL, выпуск динамических сборок и т.д.). С помощью этой книги вы сможете уверенно использовать язык C# на практике и хорошо ориентироваться в мире .NET.

Основные темы книги

- Возможности языка C# 9 и обновления в записях, неизменяемых классах, средствах доступа только для инициализации, операторах верхнего уровня, сопоставлении с образцом и т.д.
- Начало работы с веб-приложениями и веб-службами ASP.NET Core
- Использование Entity Framework Core для построения реальных приложений, управляющих данными, с расширенным охватом нововведений этой версии
- Разработка приложений с помощью C# и современных инфраструктур для служб, веб-сети и интеллектуальных клиентов
- Философия, лежащая в основе .NET
- Новые средства .NET 5, включая однофайловые приложения, уменьшенные образы контейнеров, поддержку Windows ARM64 и многое другое
- Разработка настольных приложений Windows в .NET 5 с использованием Windows Presentation Foundation
- Улучшение показателей производительности благодаря обновлениям ASP.NET Core, Entity Framework Core и внутренних механизмов, таких как сборка мусора, System.Text.Json и оптимизация размера контейнера

Категория: языки программирования/C#

Предмет рассмотрения: C# 9

Уровень: для пользователей средней и высокой квалификации

ISBN 978-617-7987-82-5

22007



9 786177 987825

Apress®
www.apress.com

Комп'ютерне видавництво
"Діалектика"
www.dialektika.com